Core DOM

A Formal Model of the Document Object Model

Achim D. Brucker Michael Herzberg

January 7, 2019

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

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].

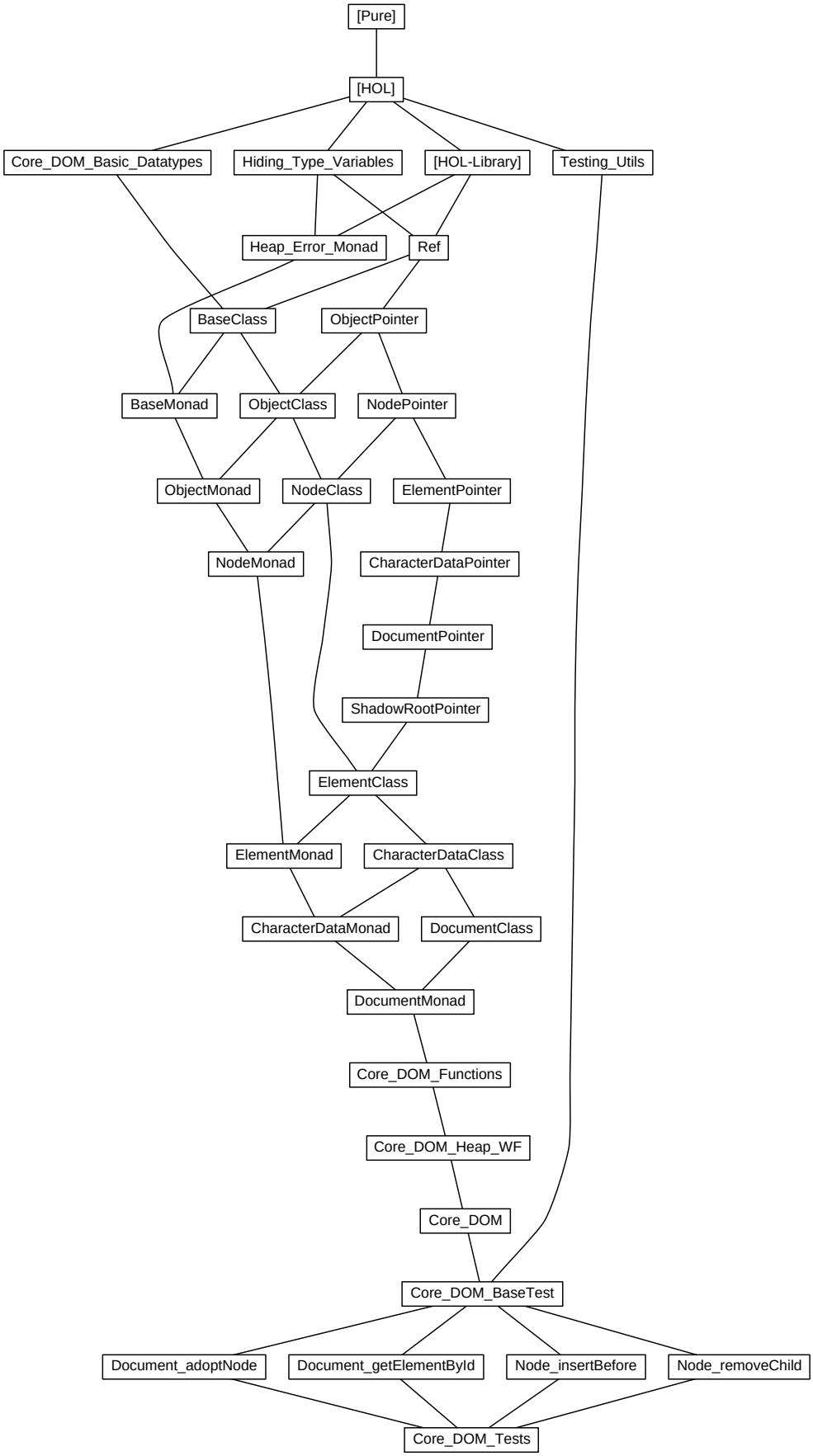

Figure 1.1: The Dependency Graph of the Isabelle Theories.

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
```

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'"*
  by *(auto simp add: returns_result_def split: sum.splits)*

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)"

lemma *returns_heap_eq [elim]: "h ⊢ f →h h' ⟹ h ⊢ f →h h'' ⟹ h' = h''"*
by *(auto simp add: returns_heap_def split: sum.splits)*

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"*
by *(auto simp add: is_OK_def returns_result_def split: sum.splits)*

lemma *is_OK_returns_result_E [elim]:*
assumes *"h ⊢ ok f"*
obtains *x* **where** *"h ⊢ f →r x"*
using *assms* **by** *(auto simp add: is_OK_def returns_result_def split: sum.splits)*

lemma *is_OK_returns_heap_I [intro]: "h ⊢ f →h h' ⟹ h ⊢ ok f"*
by *(auto simp add: is_OK_def returns_heap_def split: sum.splits)*

lemma *is_OK_returns_heap_E [elim]:*
assumes *"h ⊢ ok f"*
obtains *h'* **where** *"h ⊢ f →h h'"*
using *assms* **by** *(auto simp add: is_OK_def returns_heap_def split: sum.splits)*

lemma *select_result_I:*
assumes *"h ⊢ ok f"*
and *"⋀x. h ⊢ f →r x ⟹ P x"*
shows *"P |h ⊢ f|r"*
using *assms*
by *(auto simp add: is_OK_def returns_result_def split: sum.splits)*

lemma *select_result_I2 [simp]:*
assumes *"h ⊢ f →r x"*
shows *"|h ⊢ f|r = x"*
using *assms*
by *(auto simp add: is_OK_def returns_result_def split: sum.splits)*

lemma *returns_result_select_result [simp]:*
assumes *"h ⊢ ok f"*
shows *"h ⊢ f →r |h ⊢ f|r"*
using *assms*
by *(simp add: select_result_I)*

lemma *select_result_E:*
assumes *"P |h ⊢ f|r"* **and** *"h ⊢ ok f"*
obtains *x* **where** *"h ⊢ f →r x"* **and** *"P x"*
using *assms*
by *(auto simp add: is_OK_def returns_result_def split: sum.splits)*

lemma *select_result_eq: "(⋀x .h ⊢ f →r x = h' ⊢ f →r x) ⟹ |h ⊢ f|r = |h' ⊢ f|r"*
by *(metis (no_types, lifting) is_OK_def old.sum.simps(6) select_result.elims*
select_result_I select_result_I2)

definition *error :: "'e ⇒ ('heap, 'e, 'result) prog"*
where

```
    "error exception = Prog (λh. Inl exception)"

lemma error_bind [iff]: "(error e ⤜ g) = error e"
  unfolding error_def bind_def by auto

lemma error_returns_result [simp]: "¬ (h ⊢ error e →r y)"
  unfolding returns_result_def error_def execute_def by auto

lemma error_returns_heap [simp]: "¬ (h ⊢ error e →h h')"
  unfolding returns_heap_def error_def execute_def by auto

lemma error_returns_error [simp]: "h ⊢ error e →e e"
  unfolding returns_error_def error_def execute_def by auto

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

lemma return_ok [simp]: "h ⊢ ok (return x)"
  by(simp add: return_def is_OK_def execute_def)

lemma return_bind [iff]: "(return x ⤜ g) = g x"
  unfolding return_def bind_def by auto

lemma return_id [simp]: "f ⤜ return = f"
  by (induct f) (auto simp add: return_def bind_def split: sum.splits prod.splits)

lemma return_returns_result [iff]: "(h ⊢ return x →r y) = (x = y)"
  unfolding returns_result_def return_def execute_def by auto

lemma return_returns_heap [iff]: "(h ⊢ return x →h h') = (h = h')"
  unfolding returns_heap_def return_def execute_def by auto

lemma return_returns_error [iff]: "¬ h ⊢ return x →e e"
  unfolding returns_error_def execute_def return_def by auto

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

lemma noop_returns_heap [simp]: "h ⊢ noop →h h' ⟷ h = h'"
  by(simp add: noop_def)

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

lemma get_heap_ok [simp]: "h ⊢ ok (get_heap)"
  by (simp add: get_heap_def execute_def is_OK_def return_def)

lemma get_heap_returns_result [simp]: "(h ⊢ get_heap ⤜ (λh'. f h') →r x) = (h ⊢ f h →r x)"
  by(simp add: get_heap_def returns_result_def bind_def return_def execute_def)

lemma get_heap_returns_heap [simp]: "(h ⊢ get_heap ⤜ (λh'. f h') →h h'') = (h ⊢ f h →h h'')"
  by(simp add: get_heap_def returns_heap_def bind_def return_def execute_def)

lemma get_heap_is_OK [simp]: "(h ⊢ ok (get_heap ⤜ (λh'. f h'))) = (h ⊢ ok (f h))"
  by(auto simp add: get_heap_def is_OK_def bind_def return_def execute_def)

lemma get_heap_E [elim]: "(h ⊢ get_heap →r x) ⟹ x = h"
  by(simp add: get_heap_def returns_result_def return_def execute_def)

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' →h h'') = (h'' = h')"*
by *(simp add: return_heap_def returns_heap_def return_def execute_def)*

lemma *return_heap_returns_result [simp]: "h ⊢ return_heap h' →r ()"*
by *(simp add: return_heap_def execute_def returns_result_def return_def)*

2.2.3 Pure Heaps

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

lemma *return_pure [simp]: "pure (return x) h"*
by *(simp add: pure_def return_def is_OK_def returns_heap_def execute_def)*

lemma *error_pure [simp]: "pure (error e) h"*
by *(simp add: pure_def error_def is_OK_def returns_heap_def execute_def)*

lemma *noop_pure [simp]: "pure (noop) h"*
by *(simp add: noop_def)*

lemma *get_pure [simp]: "pure get_heap h"*
by *(simp add: pure_def get_heap_def is_OK_def returns_heap_def return_def execute_def)*

lemma *pure_returns_heap_eq:*
"h ⊢ f →h h' ⟹ pure f h ⟹ h = h'"
by *(meson pure_def is_OK_returns_heap_I returns_heap_eq)*

lemma *pure_eq_iff:*
"(∀h' x. h ⊢ f →r x ⟶ h ⊢ f →h h' ⟶ h = h') ⟷ pure f h"
by *(auto simp add: pure_def)*

2.2.4 Bind

lemma *bind_assoc [simp]:*
"((bind f g) ⤜ h) = (f ⤜ (λx. (g x ⤜ h)))"
by *(auto simp add: bind_def split: sum.splits)*

lemma *bind_returns_result_E:*
assumes *"h ⊢ f ⤜ g →r y"*
obtains *x h'* **where** *"h ⊢ f →r x"* **and** *"h ⊢ f →h h'"* **and** *"h' ⊢ g x →r y"*
using *assms* **by** *(auto simp add: bind_def returns_result_def returns_heap_def execute_def split: sum.splits)*

lemma *bind_returns_result_E2:*
assumes *"h ⊢ f ⤜ g →r y"* **and** *"pure f h"*
obtains *x* **where** *"h ⊢ f →r x"* **and** *"h ⊢ g x →r y"*
using *assms pure_returns_heap_eq bind_returns_result_E* **by** *metis*

lemma *bind_returns_result_E3:*
assumes *"h ⊢ f ⤜ g →r y"* **and** *"h ⊢ f →r x"* **and** *"pure f h"*
shows *"h ⊢ g x →r y"*
using *assms returns_result_eq bind_returns_result_E2* **by** *metis*

lemma *bind_returns_result_E4:*
assumes *"h ⊢ f ⤜ g →r y"* **and** *"h ⊢ f →r x"*
obtains *h'* **where** *"h ⊢ f →h h'"* **and** *"h' ⊢ g x →r y"*
using *assms returns_result_eq bind_returns_result_E* **by** *metis*

lemma *bind_returns_heap_E:*
assumes *"h ⊢ f ⤜ g →h h''"*

```
  obtains x h' where "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ g x →h h''"
  using assms by(auto simp add: bind_def returns_result_def returns_heap_def execute_def
      split: sum.splits)

lemma bind_returns_heap_E2 [elim]:
  assumes "h ⊢ f ⤜ g →h h'" and "pure f h"
  obtains x where "h ⊢ f →r x" and "h ⊢ g x →h h'"
  using assms pure_returns_heap_eq by (fastforce elim: bind_returns_heap_E)

lemma bind_returns_heap_E3 [elim]:
  assumes "h ⊢ f ⤜ g →h h'" and "h ⊢ f →r x" and "pure f h"
  shows "h ⊢ g x →h h'"
  using assms pure_returns_heap_eq returns_result_eq by (fastforce elim: bind_returns_heap_E)

lemma bind_returns_heap_E4:
  assumes "h ⊢ f ⤜ g →h h''" and "h ⊢ f →h h'"
  obtains x where "h ⊢ f →r x" and "h' ⊢ g x →h h''"
  using assms
  by (metis bind_returns_heap_E returns_heap_eq)

lemma bind_returns_error_I [intro]:
  assumes "h ⊢ f →e e"
  shows "h ⊢ f ⤜ g →e e"
  using assms
  by(auto simp add: returns_error_def bind_def execute_def split: sum.splits)

lemma bind_returns_error_I3:
  assumes "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ g x →e e"
  shows "h ⊢ f ⤜ g →e e"
  using assms
  by(auto simp add: returns_error_def bind_def execute_def returns_heap_def returns_result_def
      split: sum.splits)

lemma bind_returns_error_I2 [intro]:
  assumes "pure f h" and "h ⊢ f →r x" and "h ⊢ g x →e e"
  shows "h ⊢ f ⤜ g →e e"
  using assms
  by (meson bind_returns_error_I3 is_OK_returns_result_I pure_def)

lemma bind_is_OK_E [elim]:
  assumes "h ⊢ ok (f ⤜ g)"
  obtains x h' where "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ ok (g x)"
  using assms
  by(auto simp add: bind_def returns_result_def returns_heap_def is_OK_def execute_def
      split: sum.splits)

lemma bind_is_OK_E2:
  assumes "h ⊢ ok (f ⤜ g)" and "h ⊢ f →r x"
  obtains h' where "h ⊢ f →h h'" and "h' ⊢ ok (g x)"
  using assms
  by(auto simp add: bind_def returns_result_def returns_heap_def is_OK_def execute_def
      split: sum.splits)

lemma bind_returns_result_I [intro]:
  assumes "h ⊢ f →r x" and "h ⊢ f →h h'" and "h' ⊢ g x →r y"
  shows "h ⊢ f ⤜ g →r y"
  using assms
  by(auto simp add: bind_def returns_result_def returns_heap_def execute_def
      split: sum.splits)

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"
```

```
  using assms
  by (meson bind_returns_result_I pure_def is_OK_returns_result_I)

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"
  using assms by auto

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''"
  using assms
  by(auto simp add: bind_def returns_result_def returns_heap_def execute_def
      split: sum.splits)

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''"
  using assms
  by (meson bind_returns_heap_I is_OK_returns_heap_I is_OK_returns_result_E)

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)"
  by (meson assms(1) assms(2) assms(3) bind_returns_heap_I is_OK_returns_heap_E
      is_OK_returns_heap_I)

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)"
  using assms by blast

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)"
  using assms by blast

lemma bind_pure_I:
  assumes "pure f h" and "⋀x. h ⊢ f →r x ⟹ pure (g x) h"
  shows "pure (f ⤜ g) h"
  using assms
  by (metis bind_returns_heap_E2 pure_def pure_returns_heap_eq is_OK_returns_heap_E)

lemma pure_pure:
  assumes "h ⊢ ok f" and "pure f h"
  shows "h ⊢ f →h h"
  using assms returns_heap_eq
  unfolding pure_def
  by auto

lemma bind_returns_error_eq:
  assumes "h ⊢ f →e e"
    and "h ⊢ g →e e"
  shows "h ⊢ f = h ⊢ g"
  using assms
  by(auto simp add: returns_error_def split: sum.splits)
```

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)"
  apply(induct xs)
  by (simp_all add: bind_is_OK_I2 bind_is_OK_pure_I)

lemma map_M_pure_I : "⋀h. (⋀x. x ∈ set xs ⟹ pure (f x) h) ⟹ pure (map_M f xs) h"
  apply(induct xs)
   apply(simp)
  by(auto intro!: bind_pure_I)

lemma map_M_pure_E :
  assumes "h ⊢ map_M g xs →r ys" and "x ∈ set xs" and "⋀x h. x ∈ set xs ⟹ pure (g x) h"
  obtains y where "h ⊢ g x →r y" and "y ∈ set ys"
  apply(insert assms, induct xs arbitrary: ys)
   apply(simp)
  apply(auto elim!: bind_returns_result_E)[1]
  by (metis (full_types) pure_returns_heap_eq)

lemma map_M_pure_E2:
  assumes "h ⊢ map_M g xs →r ys" and "y ∈ set ys" and "⋀x h. x ∈ set xs ⟹ pure (g x) h"
  obtains x where "h ⊢ g x →r y" and "x ∈ set xs"
  apply(insert assms, induct xs arbitrary: ys)
   apply(simp)
  apply(auto elim!: bind_returns_result_E)[1]
  by (metis (full_types) pure_returns_heap_eq)
```

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"
  apply(induct xs)
  by(auto intro!: bind_pure_I)
```

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)"
  apply(induct xs)
  by(auto intro: bind_pure_I)
```

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"
  apply(induct xs)
  by(auto intro!: bind_pure_I)

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)"
  apply(induct xs)
   apply(simp)
  by(auto intro!: bind_is_OK_pure_I)

lemma filter_M_not_more_elements:
  assumes "h ⊢ filter_M P xs →r ys" and "⋀x. x ∈ set xs ⟹ pure (P x) h" and "x ∈ set ys"
  shows "x ∈ set xs"
  apply(insert assms, induct xs arbitrary: ys)
  by(auto elim!: bind_returns_result_E2 split: if_splits intro!: set_ConsD)

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"
  apply(insert assms, induct xs arbitrary: ys)
   apply(simp)
  apply(auto elim!: bind_returns_result_E2)[1]
  by (metis returns_result_eq)

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"
  apply(insert assms, induct xs arbitrary: ys)
  by(auto elim!: bind_returns_result_E2 split: if_splits intro!: set_ConsD)

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 []"
  using assms
  apply(induct xs)
  by(auto intro!: bind_pure_returns_result_I)

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 -
  assume 1: "h ⊢ filter_M P xs →r ys" and 2: "h' ⊢ filter_M P xs →r ys'"
    and 3: "(⋀x. pure (P x) h)" and "(⋀x. pure (P x) h')"
    and 4: "∀b. ∀x∈set xs. h ⊢ P x →r True ⟶ h' ⊢ P x →r b ⟶ b"
  have h1: "∀x ∈ set xs. h' ⊢ ok (P x)"
    using 2 3 ‹(⋀x. pure (P x) h')›
    apply(induct xs arbitrary: ys')
    by(auto elim!: bind_returns_result_E2)
  then have 5: "∀x∈set xs. h ⊢ P x →r True ⟶ h' ⊢ P x →r True"
    using 4
    apply(auto)[1]
    by (metis is_OK_returns_result_E)
  show ?thesis
    using 1 2 3 5 ‹(⋀x. pure (P x) h')›
```

```
    apply(induct xs arbitrary: ys ys')
     apply(auto)[1]
    apply(auto elim!: bind_returns_result_E2 split: if_splits)[1]
          apply auto[1]
         apply auto[1]
        apply(metis returns_result_eq)
       apply auto[1]
      apply auto[1]
     apply auto[1]
    by(auto)
qed

lemma filter_M_subset: "h ⊢ filter_M P xs →r ys ⟹ set ys ⊆ set xs"
  apply(induct xs arbitrary: h ys)
   apply(auto)[1]
  apply(auto elim!: bind_returns_result_E split: if_splits)[1]
   apply blast
  by blast

lemma filter_M_distinct: "h ⊢ filter_M P xs →r ys ⟹ distinct xs ⟹ distinct ys"
  apply(induct xs arbitrary: h ys)
   apply(auto)[1]
  using filter_M_subset
  apply(auto elim!: bind_returns_result_E)[1]
  by fastforce

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"
  apply(induct xs arbitrary: ys)
  by(auto elim!: bind_returns_result_E2)

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"
  apply(induct xs arbitrary: ys)
  by(auto elim!: bind_returns_result_E2 intro!: bind_pure_returns_result_I)

lemma filter_ex1: "∃!x ∈ set xs. P x ⟹ P x ⟹ x ∈ set xs ⟹ distinct xs
                 ⟹ filter P xs = [x]"
  apply(auto)[1]
  apply(induct xs)
   apply(auto)[1]
  apply(auto)[1]
  using filter_empty_conv by fastforce

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 -
  have *: "∃!x ∈ set xs. |h ⊢ P x|r"
    apply(insert assms(1) assms(3) assms(4))
    apply(drule filter_M_filter)
     apply(simp)
    apply(auto simp add: select_result_I2)[1]
    by (metis (full_types) is_OK_returns_result_E select_result_I2)
  then show ?thesis
    apply(insert assms(1) assms(4))
    apply(drule filter_M_filter)
     apply(auto)[1]
```

```
    by (metis * assms(2) assms(5) assms(6) distinct_filter
        distinct_length_2_or_more filter_empty_conv filter_set list.exhaust
        list.set_intros(1) list.set_intros(2) member_filter select_result_I2)
qed

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"
  using assms
  apply (induct xs arbitrary: ys)
  by(auto elim!: bind_returns_result_E2 intro!: bind_pure_returns_result_I
      dest: returns_result_eq)
```

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"
  by(auto simp add: map_filter_M_def map_M_pure_I intro!: bind_pure_I)

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 →r ys" and "y ∈ set ys" and "⋀x h. x ∈ set xs ⟹ pure (f x) h"
  obtains x where "h ⊢ f x →r Some y" and "x ∈ set xs"
proof -
  obtain ys_opts ys_no_opts where
    ys_opts: "h ⊢ map_M f xs →r ys_opts" and
    ys_no_opts: "h ⊢ filter_M (λx. (return (x ≠ None)::('heap, 'e, bool) prog)) ys_opts →r ys_no_opts"
and
    ys: "h ⊢ map_M (λx. (return (the x)::('heap, 'e, 'y) prog)) ys_no_opts →r ys"
    using assms
    by(auto simp add: map_filter_M_def map_M_pure_I elim!: bind_returns_result_E2)
  have "∀y ∈ set ys_no_opts. y ≠ None"
    using ys_no_opts filter_M_holds_for_result
    by fastforce
  then have "Some y ∈ set ys_no_opts"
    using map_M_pure_E2 ys ‹y ∈ set ys›
    by (metis (no_types, lifting) option.collapse return_pure return_returns_result)
  then have "Some y ∈ set ys_opts"
    using filter_M_subset ys_no_opts by fastforce
  then show "(⋀x. h ⊢ f x →r Some y ⟹ x ∈ set xs ⟹ thesis) ⟹ thesis"
    by (metis assms(3) map_M_pure_E2 ys_opts)
qed
```

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 →h h'"
    and "h' ⊢ iterate_M ys →h h''"
```

```
  shows "h ⊢ iterate_M (xs @ ys) →h h''"
  using assms
  apply(induct "xs" arbitrary: h h'')
   apply(simp)
  apply(auto)[1]
  by (meson bind_returns_heap_E bind_returns_heap_I)
```

2.2.11 Miscellaneous Rules

```
lemma execute_bind_simp:
  assumes "h ⊢ f →r x" and "h ⊢ f →h h'"
  shows "h ⊢ f ⤜ g = h' ⊢ g x"
  using assms
  by(auto simp add: returns_result_def returns_heap_def bind_def execute_def
      split: sum.splits)

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 →r y ⟹ h ⊢ f1 →h h' ⟹ h' ⊢ g1 y = h' ⊢ g2 y"
  shows "h ⊢ (f1 ⤜ g1) = h ⊢ (f2 ⤜ g2)"
  apply(insert assms, cases "h ⊢ f1")
  by(auto simp add: bind_def returns_result_def returns_heap_def execute_def
      split: sum.splits)

lemma bind_cong_2:
  assumes "pure f h" and "pure f h'"
    and "⋀x. h ⊢ f →r x = h' ⊢ f →r x"
    and "⋀x. h ⊢ f →r x ⟹ h ⊢ g x →r y = h' ⊢ g x →r y'"
  shows "h ⊢ f ⤜ g →r y = h' ⊢ f ⤜ g →r y'"
  using assms
  by(auto intro!: bind_pure_returns_result_I elim!: bind_returns_result_E2)

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"
  by (insert assms, simp add: option.case_eq_if)
```

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 →r x ⟷ h' ⊢ f →r x)"

lemma reflp_preserved_f [simp]: "reflp (preserved f)"
  by(auto simp add: preserved_def reflp_def)
lemma transp_preserved_f [simp]: "transp (preserved f)"
  by(auto simp add: preserved_def transp_def)


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'"*
by *(auto simp add: reads_def)*

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'"*
using *assms*
by *(auto simp add: reads_def pure_pure preserved_def*
intro!: bind_pure_returns_result_I is_OK_returns_result_I
dest: pure_returns_heap_eq
elim!: bind_returns_result_E)

lemma *reads_insert_writes_set_left: "∀P ∈ S. reflp P ∧ transp P ⟹ reads {getter} f h h' ⟹ reads (insert getter S) f h h'"*
unfolding *reads_def* **by** *simp*

lemma *reads_insert_writes_set_right: "reflp getter ⟹ transp getter ⟹ reads S f h h' ⟹ reads (insert getter S) f h h'"*
unfolding *reads_def* **by** *blast*

lemma *reads_subset: "reads S f h h' ⟹ ∀P ∈ S' - S. reflp P ∧ transp P ⟹ S ⊆ S' ⟹ reads S' f h h'"*
by *(auto simp add: reads_def)*

lemma *return_reads [simp]: "reads {} (return x) h h'"*
by *(simp add: reads_def preserved_def)*

lemma *error_reads [simp]: "reads {} (error e) h h'"*
by *(simp add: reads_def preserved_def)*

lemma *noop_reads [simp]: "reads {} noop h h'"*
by *(simp add: reads_def noop_def preserved_def)*

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'"*
using *assms*
apply *(induct xs)*
by *(auto intro: reads_subset[OF return_reads] intro!: reads_bind_pure)*

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'"*
apply *(auto simp add: writes_def all_args_def)[1]*
apply *(rule exI[***where** *x="[f a]"])*
by *(auto)*

lemma *writes_singleton2 [simp]: "writes {f} f h h'"*
apply *(auto simp add: writes_def all_args_def)[1]*
apply *(rule exI[***where** *x="[f]"])*
by *(auto)*

lemma *writes_union_left_I:*
assumes *"writes S f h h'"*

```
  shows "writes (S ∪ S') f h h'"
  using assms
  by(auto simp add: writes_def)

lemma writes_union_right_I:
  assumes "writes S' f h h'"
  shows "writes (S ∪ S') f h h'"
  using assms
  by(auto simp add: writes_def)

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'"
  using assms
  by(auto simp add: writes_def)

lemma writes_subset: "writes S f h h' ⟹ S ⊆ S' ⟹ writes S' f h h'"
  by(auto simp add: writes_def)

lemma writes_error [simp]: "writes S (error e) h h'"
  by(simp add: writes_def)

lemma writes_not_ok [simp]: "¬h ⊢ ok f ⟹ writes S f h h'"
  by(auto simp add: writes_def)

lemma writes_pure [simp]:
  assumes "pure f h"
  shows "writes S f h h'"
  using assms
  apply(auto simp add: writes_def)[1]
  by (metis bot.extremum iterate_M.simps(1) list.set(1) pure_returns_heap_eq return_returns_heap)

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'"
  using assms
  apply(auto simp add: writes_def elim!: bind_returns_heap_E)[1]
  by (metis iterate_M_concat le_supI set_append)

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'"
  using assms
  by(auto simp add: writes_def elim!: bind_returns_heap_E2)

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 -
  obtain progs where "set progs ⊆ SW" and iterate: "h ⊢ iterate_M progs →h h'"
    by (meson assms(1) assms(2) writes_def)
  then have "⋀h h'. ∀prog ∈ set progs. h ⊢ prog →h h' ⟶ P h h'"
    using assms(3) by auto
  with iterate assms(4) assms(5) have "h ⊢ iterate_M progs →h h' ⟹ P h h'"
  proof(induct progs arbitrary: h)
    case Nil
```

```
    then show ?case
      using reflpE by force
  next
    case (Cons a progs)
    then show ?case
      apply(auto elim!: bind_returns_heap_E)[1]
      by (metis (full_types) transpD)
  qed
  then show ?thesis
    using assms(1) iterate by blast
qed

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 -
  obtain progs where "set progs ⊆ SW" and iterate: "h ⊢ iterate_M progs →_h h'"
    by (meson assms(2) assms(3) writes_def)
  then have "⋀h h'. ∀prog ∈ set progs. h ⊢ prog →_h h' ⟶ (∀r ∈ SR. r h h')"
    using assms(4) by blast
  with iterate have "∀r ∈ SR. r h h'"
    using writes_small_big assms(1) unfolding reads_def
    by (metis assms(2) assms(3) assms(4))
  then show ?thesis
    using assms(1)
    by (simp add: preserved_def reads_def)
qed

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"
  using reads_writes_preserved[OF assms(1) assms(2) assms(3) assms(5)] assms(4)
  by(auto simp add: preserved_def)

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"
  using reads_writes_preserved[OF assms(1) assms(2) assms(3) assms(5)] assms(4)
  by(auto simp add: preserved_def)

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
    "HOL-Library.Adhoc_Overloading"
    "../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 by(standard) (auto simp add: less_eq_sum_def less_sum_def split: sum.splits)
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 by(standard, auto simp add: less_eq_object_ptr_def less_object_ptr_def
      split: object_ptr.splits)
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
  apply(standard)
  by(auto simp add: less_eq_node_ptr_def less_node_ptr_def split: node_ptr.splits)
end

lemma node_ptr_casts_commute [simp]:
  "cast_object_ptr2node_ptr ptr = Some node_ptr ⟷ cast_node_ptr2object_ptr node_ptr = ptr"
  unfolding cast_object_ptr2node_ptr_def cast_node_ptr2object_ptr_def
  by(auto split: object_ptr.splits sum.splits)

lemma node_ptr_casts_commute2 [simp]:
  "cast_object_ptr2node_ptr (cast_node_ptr2object_ptr node_ptr) = Some node_ptr"
  by simp

lemma node_ptr_casts_commute3 [simp]:
  assumes "is_node_ptr_kind ptr"
  shows "cast_node_ptr2object_ptr (the (cast_object_ptr2node_ptr ptr)) = ptr"
  using assms
  by(auto simp add: is_node_ptr_kind_def cast_node_ptr2object_ptr_def cast_object_ptr2node_ptr_def
```

```
    split: object_ptr.splits sum.splits)

lemma is_node_ptr_kind_obtains:
  assumes "is_node_ptr_kind ptr"
  obtains node_ptr where "cast_object_ptr2node_ptr ptr = Some node_ptr"
  using assms is_node_ptr_kind_def by auto

lemma is_node_ptr_kind_none:
  assumes "¬is_node_ptr_kind ptr"
  shows "cast_object_ptr2node_ptr ptr = None"
  using assms
  unfolding is_node_ptr_kind_def cast_object_ptr2node_ptr_def
  by auto

lemma is_node_ptr_kind_cast [simp]: "is_node_ptr_kind (cast_node_ptr2object_ptr node_ptr)"
  unfolding is_node_ptr_kind_def by simp

lemma cast_node_ptr2object_ptr_inject [simp]:
  "cast_node_ptr2object_ptr x = cast_node_ptr2object_ptr y ⟷ x = y"
  by(simp add: cast_node_ptr2object_ptr_def)

lemma cast_object_ptr2node_ptr_ext_none [simp]:
  "cast_object_ptr2node_ptr (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
  by(simp add: cast_object_ptr2node_ptr_def)

lemma node_ptr_inclusion [simp]:
  "cast_node_ptr2object_ptr node_ptr ∈ cast_node_ptr2object_ptr ` node_ptrs ⟷ node_ptr ∈ node_ptrs"
  by auto
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
  apply(standard)
  by(auto simp add: less_eq_element_ptr_def less_element_ptr_def split: element_ptr.splits)
end

lemma is_element_ptr_ref [simp]: "is_element_ptr (element_ptr.Ref n)"
  by(simp add: is_element_ptr_element_ptr_def)

lemma element_ptr_casts_commute [simp]:
  "cast_node_ptr2element_ptr node_ptr = Some element_ptr ⟷ cast_element_ptr2node_ptr element_ptr = node_ptr"
  unfolding cast_node_ptr2element_ptr_def cast_element_ptr2node_ptr_def
  by(auto split: node_ptr.splits sum.splits)

lemma element_ptr_casts_commute2 [simp]:
  "(cast_node_ptr2element_ptr (cast_element_ptr2node_ptr element_ptr) = Some element_ptr)"
  by simp

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"
  using assms
  by(auto simp add: is_element_ptr_kind_def cast_element_ptr2node_ptr_def cast_node_ptr2element_ptr_def
      split: node_ptr.splits sum.splits)

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"
  by (metis assms is_element_ptr_kind_def case_optionE element_ptr_casts_commute)

lemma is_element_ptr_kind_none:
  assumes "¬is_element_ptr_kind node_ptr"
  shows "cast_node_ptr2element_ptr node_ptr = None"
  using assms
  unfolding is_element_ptr_kind_def cast_node_ptr2element_ptr_def
  by(auto split: node_ptr.splits sum.splits)

lemma is_element_ptr_kind_cast [simp]:
  "is_element_ptr_kind (cast_element_ptr2node_ptr element_ptr)"
  by (metis element_ptr_casts_commute is_element_ptr_kind_none option.distinct(1))

lemma cast_element_ptr2node_ptr_inject [simp]:
  "cast_element_ptr2node_ptr x = cast_element_ptr2node_ptr y ⟷ x = y"
  by(simp add: cast_element_ptr2node_ptr_def)

lemma cast_node_ptr2element_ptr_ext_none [simp]:
  "cast_node_ptr2element_ptr (node_ptr.Ext (Inr (Inr node_ext_ptr))) = None"
  by(simp add: cast_node_ptr2element_ptr_def)

lemma is_element_ptr_implies_kind [dest]: "is_element_ptr_node_ptr ptr ⟹ is_element_ptr_kind_node_ptr ptr"
  by(auto split: option.splits)

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
  apply(standard)
  by(auto simp add: less_eq_character_data_ptr_def less_character_data_ptr_def
      split: character_data_ptr.splits)
end

lemma is_character_data_ptr_ref [simp]: "is_character_data_ptr (character_data_ptr.Ref n)"
  by(simp add: is_character_data_ptr_character_data_ptr_def)

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)"
  unfolding cast_element_ptr2node_ptr_def cast_character_data_ptr2node_ptr_def
  by(auto)

lemma is_character_data_ptr_kind_not_element_ptr [simp]:
  "¬ is_character_data_ptr_kind (cast_element_ptr2node_ptr element_ptr)"
  unfolding is_character_data_ptr_kind_def cast_element_ptr2node_ptr_def cast_node_ptr2character_data_ptr_def
  by auto
lemma is_element_ptr_kind_not_character_data_ptr [simp]:
  "¬ is_element_ptr_kind (cast_character_data_ptr2node_ptr character_data_ptr)"
  using is_element_ptr_kind_obtains by fastforce

lemma is_character_data_ptr_kind_cast [simp]:
  "is_character_data_ptr_kind (cast_character_data_ptr2node_ptr character_data_ptr)"
  by (simp add: cast_character_data_ptr2node_ptr_def cast_node_ptr2character_data_ptr_def
      is_character_data_ptr_kind_node_ptr_def)

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"
  unfolding cast_node_ptr2character_data_ptr_def cast_character_data_ptr2node_ptr_def
  by(auto split: node_ptr.splits sum.splits)

lemma character_data_ptr_casts_commute2 [simp]:
  "(cast_node_ptr2character_data_ptr (cast_character_data_ptr2node_ptr character_data_ptr) = Some character_data_ptr)"
  by simp

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"
  using assms
  by(auto simp add: is_character_data_ptr_kind_node_ptr_def cast_node_ptr2character_data_ptr_def
      cast_character_data_ptr2node_ptr_def
      split: node_ptr.splits sum.splits)

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"
  by (metis assms is_character_data_ptr_kind_node_ptr_def case_optionE
      character_data_ptr_casts_commute)

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"
  using assms
  unfolding is_character_data_ptr_kind_node_ptr_def cast_node_ptr2character_data_ptr_def
  by(auto split: node_ptr.splits sum.splits)

lemma cast_character_data_ptr2node_ptr_inject [simp]:
  "cast_character_data_ptr2node_ptr x = cast_character_data_ptr2node_ptr y ⟷ x = y"
  by(simp add: cast_character_data_ptr2node_ptr_def)

lemma cast_node_ptr2character_data_ptr_ext_none [simp]:
  "cast_node_ptr2character_data_ptr (node_ptr.Ext (Inr (Inr node_ext_ptr))) = None"
  by(simp add: cast_node_ptr2character_data_ptr_def)

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
  apply(standard)
  by(auto simp add: less_eq_document_ptr_def less_document_ptr_def split: document_ptr.splits)
end

lemma is_document_ptr_ref [simp]: "is_document_ptr (document_ptr.Ref n)"
  by(simp add: is_document_ptr_document_ptr_def)

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"
  unfolding cast_document_ptr2object_ptr_def cast_node_ptr2object_ptr_def
  by auto

lemma document_ptr_no_node_ptr_cast [simp]:
  "¬ is_document_ptr_kind (cast_node_ptr2object_ptr node_ptr)"
  by(simp add: cast_node_ptr2object_ptr_def cast_object_ptr2document_ptr_def is_document_ptr_kind_def)
lemma node_ptr_no_document_ptr_cast [simp]:
  "¬ is_node_ptr_kind (cast_document_ptr2object_ptr document_ptr)"
  using is_node_ptr_kind_obtains by fastforce

lemma document_ptr_document_ptr_cast [simp]:
  "is_document_ptr_kind (cast_document_ptr2object_ptr document_ptr)"
  by (simp add: cast_document_ptr2object_ptr_def cast_object_ptr2document_ptr_def is_document_ptr_kind_def)
```

```
lemma document_ptr_casts_commute [simp]:
  "cast_{object_ptr2document_ptr} ptr = Some document_ptr ⟷ cast_{document_ptr2object_ptr} document_ptr = ptr"
  unfolding cast_{object_ptr2document_ptr}_def cast_{document_ptr2object_ptr}_def
  by(auto split: object_ptr.splits sum.splits)

lemma document_ptr_casts_commute2 [simp]:
  "(cast_{object_ptr2document_ptr} (cast_{document_ptr2object_ptr} document_ptr) = Some document_ptr)"
  by simp

lemma document_ptr_casts_commute3 [simp]:
  assumes "is_document_ptr_kind ptr"
  shows "cast_{document_ptr2object_ptr} (the (cast_{object_ptr2document_ptr} ptr)) = ptr"
  using assms
  by(auto simp add: is_document_ptr_kind_def cast_{document_ptr2object_ptr}_def cast_{object_ptr2document_ptr}_def
      split: object_ptr.splits sum.splits)

lemma is_document_ptr_kind_obtains:
  assumes "is_document_ptr_kind ptr"
  obtains document_ptr where "ptr = cast_{document_ptr2object_ptr} document_ptr"
  using assms is_document_ptr_kind_def
  by (metis case_optionE document_ptr_casts_commute)

lemma is_document_ptr_kind_none:
  assumes "¬is_document_ptr_kind ptr"
  shows "cast_{object_ptr2document_ptr} ptr = None"
  using assms
  unfolding is_document_ptr_kind_def cast_{object_ptr2document_ptr}_def
  by (auto split: object_ptr.splits sum.splits)

lemma cast_{document_ptr2object_ptr}_inject [simp]:
  "cast_{document_ptr2object_ptr} x = cast_{document_ptr2object_ptr} y ⟷ x = y"
  by(simp add: cast_{document_ptr2object_ptr}_def)

lemma cast_{object_ptr2document_ptr}_ext_none [simp]:
  "cast_{object_ptr2document_ptr} (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
  by(simp add: cast_{object_ptr2document_ptr}_def)

lemma is_document_ptr_kind_not_element_ptr_kind [dest]:
  "is_document_ptr_kind ptr ⟹ ¬ is_element_ptr_kind ptr"
  by(auto simp add:  split: option.splits)
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_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


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
  apply(standard)
  by(auto simp add: less_eq_shadow_root_ptr_def less_shadow_root_ptr_def
      split: shadow_root_ptr.splits)
end


lemma is_shadow_root_ptr_ref [simp]: "is_shadow_root_ptr (shadow_root_ptr.Ref n)"
  by(simp add: is_shadow_root_ptr_shadow_root_ptr_def)

lemma is_shadow_root_ptr_not_node_ptr[simp]: "¬is_shadow_root_ptr (cast_node_ptr2object_ptr node_ptr)"
  by(simp add: is_shadow_root_ptr_def cast_node_ptr2object_ptr_def cast_object_ptr2shadow_root_ptr_def)
```

```
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"
  unfolding cast_{shadow_root_ptr2object_ptr}_def cast_{node_ptr2object_ptr}_def by auto

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"
  unfolding cast_{shadow_root_ptr2object_ptr}_def cast_{document_ptr2object_ptr}_def by auto

lemma shadow_root_ptr_no_node_ptr_cast [simp]:
  "¬ is_shadow_root_ptr_kind (cast_{node_ptr2object_ptr} node_ptr)"
  by(simp add: cast_{node_ptr2object_ptr}_def cast_{object_ptr2shadow_root_ptr}_def is_shadow_root_ptr_kind_def)
lemma node_ptr_no_shadow_root_ptr_cast [simp]:
  "¬ is_node_ptr_kind (cast_{shadow_root_ptr2object_ptr} shadow_root_ptr)"
  using is_node_ptr_kind_obtains by fastforce

lemma shadow_root_ptr_no_document_ptr_cast [simp]:
  "¬ is_shadow_root_ptr_kind (cast_{document_ptr2object_ptr} document_ptr)"
  by(simp add: cast_{document_ptr2object_ptr}_def cast_{object_ptr2shadow_root_ptr}_def is_shadow_root_ptr_kind_def)
lemma document_ptr_no_shadow_root_ptr_cast [simp]:
  "¬ is_document_ptr_kind (cast_{shadow_root_ptr2object_ptr} shadow_root_ptr)"
  using is_document_ptr_kind_obtains by fastforce

lemma shadow_root_ptr_shadow_root_ptr_cast [simp]:
  "is_shadow_root_ptr_kind (cast_{shadow_root_ptr2object_ptr} shadow_root_ptr)"
  by (simp add: cast_{shadow_root_ptr2object_ptr}_def cast_{object_ptr2shadow_root_ptr}_def is_shadow_root_ptr_kind_def)

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"
  unfolding cast_{object_ptr2shadow_root_ptr}_def cast_{shadow_root_ptr2object_ptr}_def
  by(auto split: object_ptr.splits sum.splits)

lemma shadow_root_ptr_casts_commute2 [simp]:
  "(cast_{object_ptr2shadow_root_ptr} (cast_{shadow_root_ptr2object_ptr} shadow_root_ptr) = Some shadow_root_ptr)"
  by simp

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"
  using assms
  by(auto simp add: is_shadow_root_ptr_kind_def cast_{shadow_root_ptr2object_ptr}_def cast_{object_ptr2shadow_root_ptr}_def
      split: object_ptr.splits sum.splits)

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"
  using assms is_shadow_root_ptr_kind_def
  by (metis case_optionE shadow_root_ptr_casts_commute)

lemma is_shadow_root_ptr_kind_none:
  assumes "¬is_shadow_root_ptr_kind ptr"
  shows "cast_{object_ptr2shadow_root_ptr} ptr = None"
  using assms
  unfolding is_shadow_root_ptr_kind_def cast_{object_ptr2shadow_root_ptr}_def
  by (auto split: object_ptr.splits sum.splits)

lemma cast_{shadow_root_ptr2object_ptr}_inject [simp]:
  "cast_{shadow_root_ptr2object_ptr} x = cast_{shadow_root_ptr2object_ptr} y ⟷ x = y"
  by(simp add: cast_{shadow_root_ptr2object_ptr}_def)
```

lemma *cast$_{object_ptr2shadow_root_ptr}$_ext_none [simp]:*
"cast$_{object_ptr2shadow_root_ptr}$ (object_ptr.Ext (Inr (Inr (Inr object_ext_ptr)))) = None"
by *(simp add: cast$_{object_ptr2shadow_root_ptr}$_def)*

lemma *is_shadow_root_ptr_kind_simp1 [dest]: "is_document_ptr_kind ptr ⟹ ¬is_shadow_root_ptr_kind ptr"*
by *(metis document_ptr_no_shadow_root_ptr_cast shadow_root_ptr_casts_commute3)*

lemma *is_shadow_root_ptr_kind_simp2 [dest]: "is_node_ptr_kind ptr ⟹ ¬is_shadow_root_ptr_kind ptr"*
by *(metis node_ptr_no_shadow_root_ptr_cast shadow_root_ptr_casts_commute3)*

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"

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"
  by(auto simp add: object_ptr_kinds_def)

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 .
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"
  using l_type_wf_Object_axioms assms
  apply(simp add: type_wf_def get_Object_def)
  by (simp add: fmlookup_dom_iff object_ptr_kinds_def)
end

global_interpretation l_get_Object_lemmas type_wf
  by (simp add: l_get_Object_lemmas.intro l_type_wf_Object.intro)

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'"
  using assms
  unfolding put_Object_def
  by auto

lemma put_Object_put_ptrs:
```

```
  assumes "put_Object object_ptr object h = h'"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|object_ptr|}"
  using assms
  by (metis comp_apply fmdom_fmupd funion_finsert_right heap.sel object_ptr_kinds_def
      sup_bot.right_neutral put_Object_def)

lemma object_more_extend_id [simp]: "more (extend x y) = y"
  by(simp add: extend_def)

lemma object_empty [simp]: "(|nothing = (), ... = more x|) = x"
  by simp

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"
  by(simp add: a_known_ptr_def)
end
global_interpretation l_known_ptr_Object defines known_ptr = a_known_ptr .
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'"

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. ptr |∈| object_ptr_kinds h ⟶ known_ptr ptr)"

lemma known_ptrs_known_ptr:
  "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  by(simp add: a_known_ptrs_def)

lemma known_ptrs_preserved: "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
lemma known_ptrs_subset: "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
end
global_interpretation l_known_ptrs_Object known_ptr defines known_ptrs = a_known_ptrs .
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  using known_ptrs_known_ptr known_ptrs_preserved l_known_ptrs_def known_ptrs_subset by blast


lemma get_object_ptr_simp1 [simp]: "get_Object object_ptr (put_Object object_ptr object h) = Some object"
  by(simp add: get_Object_def put_Object_def)
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"
  by(simp add: get_Object_def put_Object_def)
```

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'"
  using assms
  by(auto simp add: delete_Object_def object_ptr_kinds_def split: if_splits)

lemma delete_Object_pointer_ptr_in_heap:
  assumes "delete_Object ptr h = Some h'"
  shows "ptr |∈| object_ptr_kinds h"
  using assms
  by(auto simp add: delete_Object_def object_ptr_kinds_def split: if_splits)

lemma delete_Object_ok:
  assumes "ptr |∈| object_ptr_kinds h"
  shows "delete_Object ptr h ≠ None"
  using assms
  by(auto simp add: delete_Object_def object_ptr_kinds_def split: if_splits)

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"


definition node_ptr_kinds :: "(_) heap ⇒ (_) node_ptr fset"
  where
    "node_ptr_kinds heap =
    (the |`| (cast_object_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"
  apply(auto simp add: node_ptr_kinds_def)[1]
  by force

definition cast_Object2Node :: "(_) Object ⇒ (_) Node option"
  where
    "cast_Object2Node obj = (case RObject.more obj of Inl node
    ⇒ Some (RObject.extend (RObject.truncate obj) node) | _ ⇒ None)"
adhoc_overloading cast cast_Object2Node

definition cast_Node2Object:: "(_) Node ⇒ (_) Object"
  where
    "cast_Node2Object node = (RObject.extend (RObject.truncate node) (Inl (RObject.more node)))"
adhoc_overloading cast cast_Node2Object

definition is_node_kind :: "(_) Object ⇒ bool"
  where
    "is_node_kind ptr ⟷ cast_Object2Node ptr ≠ None"

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

locale l_type_wf_def_Node
begin
definition a_type_wf :: "(_) heap ⇒ bool"
  where
    "a_type_wf h = (ObjectClass.type_wf h
                  ∧ (∀node_ptr. node_ptr |∈| node_ptr_kinds h
                                    ⟶ get_Node node_ptr h ≠ None))"
end
global_interpretation l_type_wf_def_Node defines type_wf = a_type_wf .
lemmas type_wf_defs = a_type_wf_def

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

sublocale l_type_wf_Node ⊆ l_type_wf_Object
  apply(unfold_locales)
  using ObjectClass.a_type_wf_def by auto

locale l_get_Node_lemmas = l_type_wf_Node
begin
sublocale l_get_Object_lemmas by unfold_locales
lemma get_Node_type_wf:
  assumes "type_wf h"
  shows "node_ptr |∈| node_ptr_kinds h ⟷ get_Node node_ptr h ≠ None"
  using l_type_wf_Node_axioms assms
  apply(simp add: type_wf_defs get_Node_def l_type_wf_Node_def)
  by (metis (mono_tags, lifting) bind_eq_None_conv ffmember_filter fimage_eqI
      is_node_ptr_kind_cast get_Object_type_wf local.l_type_wf_Object_axioms
      node_ptr_casts_commute2 node_ptr_kinds_def option.sel option.simps(3))
end

global_interpretation l_get_Node_lemmas type_wf
  by unfold_locales

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'"
  using assms
  unfolding put_Node_def node_ptr_kinds_def
  by (metis ffmember_filter fimage_eqI is_node_ptr_kind_cast node_ptr_casts_commute2
      option.sel put_Object_ptr_in_heap)

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|}"
  using assms
  by (simp add: put_Node_def put_Object_put_ptrs)

lemma node_ptr_kinds_commutes [simp]:
  "cast node_ptr |∈| object_ptr_kinds h ⟷ node_ptr |∈| node_ptr_kinds h"
  apply(auto simp add: node_ptr_kinds_def split: option.splits)[1]
  by (metis (no_types, lifting) ffmember_filter fimage_eqI fset.map_comp
      is_node_ptr_kind_none node_ptr_casts_commute2
      option.distinct(1) option.sel)

lemma node_empty [simp]:
  "(|RObject.nothing = (), RNode.nothing = (), ... = RNode.more node|) = node"
  by simp

lemma cast_Node2Object_inject [simp]: "cast_Node2Object x = cast_Node2Object y ⟷ x = y"
  apply(simp add: cast_Node2Object_def RObject.extend_def)
  by (metis (full_types) RObject.surjective old.unit.exhaust)

lemma cast_Object2Node_none [simp]:
  "cast_Object2Node obj = None ⟷ ¬ (∃node. cast_Node2Object node = obj)"
  apply(auto simp add: cast_Object2Node_def cast_Node2Object_def RObject.extend_def split: sum.splits)[1]
  by (metis (full_types) RObject.select_convs(2) RObject.surjective old.unit.exhaust)

lemma cast_Object2Node_some [simp]: "cast_Object2Node obj = Some node ⟷ cast node = obj"
  by(auto simp add: cast_Object2Node_def cast_Node2Object_def RObject.extend_def split: sum.splits)

lemma cast_Object2Node_inv [simp]: "cast_Object2Node (cast_Node2Object node) = Some node"
  by simp

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 .
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. ptr |∈| object_ptr_kinds h ⟶ known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  by(simp add: a_known_ptrs_def)

lemma known_ptrs_preserved: "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
```

```
lemma known_ptrs_subset: "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
end
global_interpretation l_known_ptrs_Node known_ptr defines known_ptrs = a_known_ptrs .
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  using known_ptrs_known_ptr known_ptrs_preserved l_known_ptrs_def known_ptrs_subset by blast

lemma get_node_ptr_simp1 [simp]: "get_Node node_ptr (put_Node node_ptr node h) = Some node"
  by(auto simp add: get_Node_def put_Node_def)
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"
  by(auto simp add: get_Node_def put_Node_def)

end
```

4.4 Element (ElementClass)

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

```
theory ElementClass
  imports
    NodeClass
    "../pointers/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_type = DOMString
record ('node_ptr, 'element_ptr, 'character_data_ptr, 'shadow_root_ptr) RElement = RNode +
  nothing :: unit
  tag_type :: tag_type
  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"

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_kind
h"
  apply(auto simp add: element_ptr_kinds_def)[1]
  by force

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"
  apply(auto simp add: node_ptr_kinds_def element_ptr_kinds_def)[1]
  by (metis (no_types, lifting) element_ptr_casts_commute2 ffmember_filter fimage_eqI
      fset.map_comp is_element_ptr_kind_none node_ptr_casts_commute3
      node_ptr_kinds_commutes node_ptr_kinds_def option.sel option.simps(3))
```

```
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. element_ptr |∈| element_ptr_kinds h
                                          ⟶ get_Element element_ptr h ≠ None))"
end
global_interpretation l_type_wf_def_Element defines type_wf = a_type_wf .
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
  apply(unfold_locales)
  using NodeClass.a_type_wf_def
  by (meson ElementClass.a_type_wf_def l_type_wf_Element_axioms l_type_wf_Element_def)

locale l_get_Element_lemmas = l_type_wf_Element
begin
sublocale l_get_Node_lemmas by unfold_locales

lemma get_Element_type_wf:
  assumes "type_wf h"
  shows "element_ptr |∈| element_ptr_kinds h ⟷ get_Element element_ptr h ≠ None"
  using l_type_wf_Element_axioms assms
  apply(simp add: type_wf_defs get_Element_def l_type_wf_Element_def)
  by (metis NodeClass.get_Node_type_wf bind_eq_None_conv element_ptr_kinds_commutes
      option.distinct(1))
end

global_interpretation l_get_Element_lemmas type_wf
  by unfold_locales

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'"
  using assms
  unfolding put_Element_def element_ptr_kinds_def
  by (metis element_ptr_kinds_commutes element_ptr_kinds_def put_Node_ptr_in_heap)

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|}"
  using assms
  by (simp add: put_Element_def put_Node_put_ptrs)



lemma cast_Element2Node_inject [simp]:
  "cast_Element2Node x = cast_Element2Node y ⟷ x = y"
  apply(simp add: cast_Element2Node_def RObject.extend_def RNode.extend_def)
  by (metis (full_types) RNode.surjective old.unit.exhaust)
```

```
lemma cast_Node2Element_none [simp]:
  "cast_Node2Element node = None ⟷ ¬ (∃ element. cast_Element2Node element = node)"
  apply(auto simp add: cast_Node2Element_def cast_Element2Node_def RObject.extend_def RNode.extend_def
      split: sum.splits)[1]
  by (metis (full_types) RNode.select_convs(2) RNode.surjective old.unit.exhaust)

lemma cast_Node2Element_some [simp]:
  "cast_Node2Element node = Some element ⟷ cast_Element2Node element = node"
  by(auto simp add: cast_Node2Element_def cast_Element2Node_def RObject.extend_def RNode.extend_def
      split: sum.splits)

lemma cast_Node2Element_inv [simp]: "cast_Node2Element (cast_Element2Node element) = Some element"
  by simp

lemma get_elment_ptr_simp1 [simp]:
  "get_Element element_ptr (put_Element element_ptr element h) = Some element"
  by(auto simp add: get_Element_def put_Element_def)
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"
  by(auto simp add: get_Element_def put_Element_def)


abbreviation "create_element_obj tag_type_arg child_nodes_arg attrs_arg shadow_root_opt_arg
  ≡ (| RObject.nothing = (), RNode.nothing = (), RElement.nothing = (),
      tag_type = tag_type_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
    "new_Element 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'"
  using assms
  unfolding new_Element_def Let_def
  using put_Element_ptr_in_heap by blast

lemma new_element_ptr_new:
  "element_ptr.Ref (Suc (fMax (finsert 0 (element_ptr.the_ref |`| element_ptrs h)))) |∉| element_ptrs h"
  by (metis Suc_n_not_le_n element_ptr.sel(1) fMax_ge fimage_finsert finsertI1 finsertI2 set_finsert)

lemma new_Element_ptr_not_in_heap:
  assumes "new_Element h = (new_element_ptr, h')"
  shows "new_element_ptr |∉| element_ptr_kinds h"
  using assms
  unfolding new_Element_def
  by (metis Pair_inject element_ptrs_def ffmember_filter new_element_ptr_new is_element_ptr_ref)

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|}"
  using assms
  by (metis Pair_inject new_Element_def put_Element_put_ptrs)

lemma new_Element_is_element_ptr:
  assumes "new_Element h = (new_element_ptr, h')"
```

```
  shows "is_element_ptr new_element_ptr"
  using assms
  by(auto simp add: new_Element_def Let_def)

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'"
  using assms
  by(auto simp add: new_Element_def Let_def put_Element_def put_Node_def)

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'"
  using assms
  by(auto simp add: new_Element_def Let_def put_Element_def)

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'"
  using assms
  by(auto simp add: new_Element_def Let_def)


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"
  by(simp add: a_known_ptr_def)
end
global_interpretation l_known_ptr_Element defines known_ptr = a_known_ptr .
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. ptr |∈| object_ptr_kinds h ⟶ known_ptr ptr)"

lemma known_ptrs_known_ptr:
  "ptr |∈| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ known_ptr ptr"
  by(simp add: a_known_ptrs_def)

lemma known_ptrs_preserved: "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
lemma known_ptrs_subset: "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs
h'"
  by(auto simp add: a_known_ptrs_def)
end
global_interpretation l_known_ptrs_Element known_ptr defines known_ptrs = a_known_ptrs .
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  using known_ptrs_known_ptr known_ptrs_preserved l_known_ptrs_def known_ptrs_subset by blast

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"


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"
  apply(auto simp add: character_data_ptr_kinds_def)[1]
  by force

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"
  apply(auto simp add: character_data_ptr_kinds_def)[1]
  by (metis character_data_ptr_casts_commute2 comp_eq_dest_lhs ffmember_filter fimage_eqI
      is_character_data_ptr_kind_none
      option.distinct(1) option.sel)

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. character_data_ptr |∈| 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 .
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
  apply(unfold_locales)
  using ElementClass.a_type_wf_def
  by (meson CharacterDataClass.a_type_wf_def l_type_wf_CharacterData_axioms l_type_wf_CharacterData_def)
```

```
locale l_get_CharacterData_lemmas = l_type_wf_CharacterData
begin
sublocale l_get_Element_lemmas by unfold_locales

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"
  using l_type_wf_CharacterData_axioms assms
  apply(simp add: type_wf_defs get_CharacterData_def l_type_wf_CharacterData_def)
  by (metis NodeClass.get_Node_type_wf bind_eq_None_conv character_data_ptr_kinds_commutes
      l_type_wf_Node_def local.l_type_wf_Node_axioms option.distinct(1))
end

global_interpretation l_get_CharacterData_lemmas type_wf
  by unfold_locales

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'"
  using assms put_Node_ptr_in_heap
  unfolding put_CharacterData_def character_data_ptr_kinds_def
  by (metis character_data_ptr_kinds_commutes character_data_ptr_kinds_def put_Node_ptr_in_heap)

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|}"
  using assms
  by (simp add: put_CharacterData_def put_Node_put_ptrs)


lemma cast_CharacterData2Node_inject [simp]: "cast_CharacterData2Node x = cast_CharacterData2Node y ⟷ x
= y"
  apply(simp add: cast_CharacterData2Node_def RObject.extend_def RNode.extend_def)
  by (metis (full_types) RNode.surjective old.unit.exhaust)

lemma cast_Node2CharacterData_none [simp]:
  "cast_Node2CharacterData node = None ⟷ ¬ (∃ character_data. cast_CharacterData2Node character_data = node)"
  apply(auto simp add: cast_Node2CharacterData_def cast_CharacterData2Node_def RObject.extend_def RNode.extend_def

      split: sum.splits)[1]
  by (metis (full_types) RNode.select_convs(2) RNode.surjective old.unit.exhaust)

lemma cast_Node2CharacterData_some [simp]:
  "cast_Node2CharacterData node = Some character_data ⟷ cast_CharacterData2Node character_data = node"
  by(auto simp add: cast_Node2CharacterData_def cast_CharacterData2Node_def RObject.extend_def RNode.extend_def

      split: sum.splits)

lemma cast_Node2CharacterData_inv [simp]:
  "cast_Node2CharacterData (cast_CharacterData2Node character_data) = Some character_data"
  by simp

lemma cast_element_not_character_data [simp]:
  "(cast_Element2Node element ≠ cast_CharacterData2Node character_data)"
  "(cast_CharacterData2Node character_data ≠ cast_Element2Node element)"
```

```
  by(auto simp add: cast_CharacterData2Node_def cast_Element2Node_def RNode.extend_def)

lemma get_CharacterData_simp1 [simp]:
  "get_CharacterData character_data_ptr (put_CharacterData character_data_ptr character_data h)
      = Some character_data"
  by(auto simp add: get_CharacterData_def put_CharacterData_def)
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"
  by(auto simp add: get_CharacterData_def put_CharacterData_def)

lemma get_CharacterData_simp3 [simp]:
  "get_Element element_ptr (put_CharacterData character_data_ptr f h) = get_Element element_ptr h"
  by(auto simp add: get_Element_def put_CharacterData_def)
lemma get_CharacterData_simp4 [simp]:
  "get_CharacterData element_ptr (put_Element character_data_ptr f h) = get_CharacterData element_ptr h"
  by(auto simp add: get_CharacterData_def put_Element_def)

lemma new_Element_get_CharacterData [simp]:
  assumes "new_Element h = (new_element_ptr, h')"
  shows "get_CharacterData ptr h = get_CharacterData ptr h'"
  using assms
  by(auto simp add: new_Element_def Let_def)



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

definition new_CharacterData :: "(_) heap ⇒ ((_) character_data_ptr × (_) heap)"
  where
    "new_CharacterData 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 new_CharacterData_ptr_in_heap:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "new_character_data_ptr |∈| character_data_ptr_kinds h'"
  using assms
  unfolding new_CharacterData_def Let_def
  using put_CharacterData_ptr_in_heap by blast

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"
  by (metis Suc_n_not_le_n character_data_ptr.sel(1) fMax_ge fimage_finsert finsertI1 finsertI2 set_finsert)

lemma new_CharacterData_ptr_not_in_heap:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "new_character_data_ptr |∉| character_data_ptr_kinds h"
  using assms
  unfolding new_CharacterData_def
  by (metis Pair_inject character_data_ptrs_def fMax_finsert fempty_iff ffmember_filter fimage_is_fempty
is_character_data_ptr_ref max_0L new_character_data_ptr_new)

lemma new_CharacterData_new_ptr:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
  using assms
  by (metis Pair_inject new_CharacterData_def put_CharacterData_put_ptrs)
```

```
lemma new_CharacterData_is_character_data_ptr:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "is_character_data_ptr new_character_data_ptr"
  using assms
  by(auto simp add: new_CharacterData_def Let_def)

lemma new_CharacterData_get_Object [simp]:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  assumes "ptr ≠ cast new_character_data_ptr"
  shows "get_Object ptr h = get_Object ptr h'"
  using assms
  by(auto simp add: new_CharacterData_def Let_def put_CharacterData_def put_Node_def)

lemma new_CharacterData_get_Node [simp]:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  assumes "ptr ≠ cast new_character_data_ptr"
  shows "get_Node ptr h = get_Node ptr h'"
  using assms
  by(auto simp add: new_CharacterData_def Let_def put_CharacterData_def)

lemma new_CharacterData_get_Element [simp]:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "get_Element ptr h = get_Element ptr h'"
  using assms
  by(auto simp add: new_CharacterData_def Let_def)

lemma new_CharacterData_get_CharacterData [simp]:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  assumes "ptr ≠ new_character_data_ptr"
  shows "get_CharacterData ptr h = get_CharacterData ptr h'"
  using assms
  by(auto simp add: new_CharacterData_def Let_def)


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"
  by(simp add: a_known_ptr_def)
end
global_interpretation l_known_ptr_CharacterData defines known_ptr = a_known_ptr .
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. ptr |∈| object_ptr_kinds h ⟶ known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  by(simp add: a_known_ptrs_def)

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  by(auto simp add: a_known_ptrs_def)
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  by(auto simp add: a_known_ptrs_def)
```

```
end
global_interpretation l_known_ptrs_CharacterData known_ptr defines known_ptrs = a_known_ptrs .
lemmas known_ptrs_defs = a_known_ptrs_def

lemma known_ptrs_is_l_known_ptrs: "l_known_ptrs known_ptr known_ptrs"
  using known_ptrs_known_ptr known_ptrs_preserved l_known_ptrs_def known_ptrs_subset
  by blast

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"


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"
  apply(auto simp add: document_ptr_kinds_def)[1]
  by force

lemma document_ptr_kinds_commutes [simp]:
  "cast document_ptr |∈| object_ptr_kinds h ⟷ document_ptr |∈| document_ptr_kinds h"
  apply(auto simp add: object_ptr_kinds_def document_ptr_kinds_def)[1]
  by (metis (no_types, lifting) document_ptr_casts_commute2 document_ptr_document_ptr_cast
      ffmember_filter fimage_eqI fset.map_comp option.sel)

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. document_ptr |∈| document_ptr_kinds h ⟶ get_Document document_ptr h ≠ None))"
end
global_interpretation l_type_wf_def_Document defines type_wf = a_type_wf .
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
  apply(unfold_locales)
  by (metis (full_types) type_wf_defs l_type_wf_Document_axioms l_type_wf_Document_def)

locale l_get_Document_lemmas = l_type_wf_Document
begin
sublocale l_get_CharacterData_lemmas by unfold_locales
lemma get_Document_type_wf:
  assumes "type_wf h"
  shows "document_ptr |∈| document_ptr_kinds h ⟷ get_Document document_ptr h ≠ None"
  using l_type_wf_Document_axioms assms
  apply(simp add: type_wf_defs get_Document_def l_type_wf_Document_def)
  by (metis bind_eq_None_conv document_ptr_kinds_commutes local.get_Object_type_wf
      option.distinct(1))
end

global_interpretation l_get_Document_lemmas type_wf by unfold_locales

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'"
  using assms
  unfolding put_Document_def
  by (metis document_ptr_kinds_commutes put_Object_ptr_in_heap)

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|}"
  using assms
  by (simp add: put_Document_def put_Object_put_ptrs)


lemma cast_Document2Object_inject [simp]: "cast_Document2Object x = cast_Document2Object y ⟷ x = y"
  apply(simp add: cast_Document2Object_def RObject.extend_def)
  by (metis (full_types) RObject.surjective old.unit.exhaust)

lemma cast_Object2Document_none [simp]:
  "cast_Object2Document obj = None ⟷ ¬ (∃document. cast_Document2Object document = obj)"
  apply(auto simp add: cast_Object2Document_def cast_Document2Object_def RObject.extend_def
      split: sum.splits)[1]
  by (metis (full_types) RObject.select_convs(2) RObject.surjective old.unit.exhaust)

lemma cast_Object2Document_some [simp]:
  "cast_Object2Document obj = Some document ⟷ cast document = obj"
  by(auto simp add: cast_Object2Document_def cast_Document2Object_def RObject.extend_def
      split: sum.splits)

lemma cast_Object2Document_inv [simp]: "cast_Object2Document (cast_Document2Object document) = Some document"
  by simp

lemma cast_document_not_node [simp]:
  "cast_Document2Object document ≠ cast_Node2Object node"
  "cast_Node2Object node ≠ cast_Document2Object document"
  by(auto simp add: cast_Document2Object_def cast_Node2Object_def RObject.extend_def)

lemma get_document_ptr_simp1 [simp]:
  "get_Document document_ptr (put_Document document_ptr document h) = Some document"
  by(auto simp add: get_Document_def put_Document_def)
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"
  by(auto simp add: get_Document_def put_Document_def)


lemma get_document_ptr_simp3 [simp]:
  "get_Element element_ptr (put_Document document_ptr f h) = get_Element element_ptr h"
  by(auto simp add: get_Element_def get_Node_def put_Document_def)
lemma get_document_ptr_simp4 [simp]: "get_Document document_ptr (put_Element element_ptr f h) = get_Document
document_ptr h"
  by(auto simp add: get_Document_def put_Element_def put_Node_def)
lemma get_document_ptr_simp5 [simp]:
  "get_CharacterData character_data_ptr (put_Document document_ptr f h) = get_CharacterData character_data_ptr
h"
  by(auto simp add: get_CharacterData_def get_Node_def put_Document_def)
lemma get_document_ptr_simp6 [simp]: "get_Document document_ptr (put_CharacterData character_data_ptr f h)
= get_Document document_ptr h"
  by(auto simp add: get_Document_def put_CharacterData_def put_Node_def)
```

```
lemma new_Element_get_Document [simp]:
  assumes "new_Element h = (new_element_ptr, h')"
  shows "get_Document ptr h = get_Document ptr h'"
  using assms
  by(auto simp add: new_Element_def Let_def)

lemma new_CharacterData_get_Document [simp]:
  assumes "new_CharacterData h = (new_character_data_ptr, h')"
  shows "get_Document ptr h = get_Document ptr h'"
  using assms
  by(auto simp add: new_CharacterData_def Let_def)



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 (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'"
  using assms
  unfolding new_Document_def Let_def
  using put_Document_ptr_in_heap by blast

lemma new_document_ptr_new:
  "document_ptr.Ref (Suc (fMax (finsert 0 (document_ptr.the_ref |`| document_ptrs h))))
      |∉| document_ptrs h"
  by (metis Suc_n_not_le_n document_ptr.sel(1) fMax_ge fimage_finsert finsertI1 finsertI2 set_finsert)

lemma new_Document_ptr_not_in_heap:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "new_document_ptr |∉| document_ptr_kinds h"
  using assms
  unfolding new_Document_def
  by (metis Pair_inject document_ptrs_def fMax_finsert fempty_iff ffmember_filter
      fimage_is_fempty is_document_ptr_ref max_0L new_document_ptr_new)

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|}"
  using assms
  by (metis Pair_inject new_Document_def put_Document_put_ptrs)

lemma new_Document_is_document_ptr:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "is_document_ptr new_document_ptr"
  using assms
  by(auto simp add: new_Document_def Let_def)

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'"
  using assms
  by(auto simp add: new_Document_def Let_def put_Document_def)

lemma new_Document_get_Node [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_Node ptr h = get_Node ptr h'"
  using assms
  apply(simp add: new_Document_def Let_def put_Document_def)
  by(auto simp add: get_Node_def)

lemma new_Document_get_Element [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_Element ptr h = get_Element ptr h'"
  using assms
  by(auto simp add: new_Document_def Let_def)

lemma new_Document_get_CharacterData [simp]:
  assumes "new_Document h = (new_document_ptr, h')"
  shows "get_CharacterData ptr h = get_CharacterData ptr h'"
  using assms
  by(auto simp add: new_Document_def Let_def)

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'"
  using assms
  by(auto simp add: new_Document_def Let_def)


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"
  by(simp add: a_known_ptr_def)
end
global_interpretation l_known_ptr_Document defines known_ptr = a_known_ptr .
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. ptr |∈| object_ptr_kinds h ⟶ known_ptr ptr)"

lemma known_ptrs_known_ptr: "a_known_ptrs h ⟹ ptr |∈| object_ptr_kinds h ⟹ known_ptr ptr"
  by(simp add: a_known_ptrs_def)

lemma known_ptrs_preserved:
  "object_ptr_kinds h = object_ptr_kinds h' ⟹ a_known_ptrs h = a_known_ptrs h'"
  by(auto simp add: a_known_ptrs_def)
lemma known_ptrs_subset:
  "object_ptr_kinds h' |⊆| object_ptr_kinds h ⟹ a_known_ptrs h ⟹ a_known_ptrs h'"
  by(auto simp add: a_known_ptrs_def)
end
global_interpretation l_known_ptrs_Document known_ptr defines known_ptrs = a_known_ptrs .
```

lemmas *known_ptrs_defs = a_known_ptrs_def*

lemma *known_ptrs_is_l_known_ptrs [instances]: "l_known_ptrs known_ptr known_ptrs"*
using *known_ptrs_known_ptr known_ptrs_preserved l_known_ptrs_def known_ptrs_subset* **by** *blast*

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 | SegmentationFault | HierarchyRequestError | AssertException
  | NonTerminationException | InvokeError | TypeError | DebugException nat

lemma finite_set_in [simp]: "x ∈ fset FS ⟷ x |∈| FS"
  by (meson notin_fset)

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"
  by (metis finite_fset fset_inject sorted_list_of_set(1))


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"
  by(simp add: a_ptr_kinds_M_def)

lemma ptr_kinds_M_pure [simp]: "pure a_ptr_kinds_M h"
  by (auto simp add: a_ptr_kinds_M_def intro: bind_pure_I)

lemma ptr_kinds_ptr_kinds_M [simp]: "ptr ∈ set |h ⊢ a_ptr_kinds_M|_r ⟷ ptr |∈| ptr_kinds h"
  by(simp add: a_ptr_kinds_M_def)

lemma ptr_kinds_M_ptr_kinds [simp]:
  "h ⊢ a_ptr_kinds_M →_r xa ⟷ xa = sorted_list_of_set (fset (ptr_kinds h))"
  by(auto simp add: a_ptr_kinds_M_def)
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"
  by(auto simp add: a_ptr_kinds_M_def)
```

```
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'"
  by(auto simp add: a_ptr_kinds_M_def)
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"
  by(auto simp add: a_get_M_def bind_pure_I split: option.splits)

lemma get_M_ok:
  "type_wf h ⟹ ptr |∈| ptr_kinds h ⟹ h ⊢ ok (a_get_M ptr getter)"
  apply(simp add: a_get_M_def)
  by (metis l_get_M_axioms l_get_M_def option.case_eq_if return_ok)
lemma get_M_ptr_in_heap:
  "h ⊢ ok (a_get_M ptr getter) ⟹ ptr |∈| ptr_kinds h"
  apply(simp add: a_get_M_def)
  by (metis error_returns_result is_OK_returns_result_E l_get_M_axioms l_get_M_def option.simps(4))

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)"
  by(auto simp add: a_put_M_def intro!: bind_is_OK_I2 dest: get_M_ok elim!: bind_is_OK_E)

lemma put_M_ptr_in_heap:
  "h ⊢ ok (a_put_M ptr setter v) ⟹ ptr |∈| ptr_kinds h"
  by(auto simp add: a_put_M_def intro!: bind_is_OK_I2 elim: get_M_ptr_in_heap
      dest: is_OK_returns_result_I elim!: bind_is_OK_E)

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 (unfold comp_def, rule ccpo.admissibleI, clarify)
  fix A :: "('heap ⇒ 'e + 'result × 'heap) set"
  let ?lub = "Prog (fun_lub (flat_lub (Inl e)) A)"
  fix h h2 r
  assume 1: "Complete_Partial_Order.chain (fun_ord (flat_ord (Inl e))) A"
    and 2: "∀xa∈A. ∀h h2 r. h ⊢ Prog xa = Inr (r, h2)  ⟶ P h h2 r"
    and 4: "h ⊢ Prog (fun_lub (flat_lub (Inl e)) A) = Inr (r, h2)"
  have h1:"⋀a. Complete_Partial_Order.chain (flat_ord (Inl e)) {y. ∃f∈A. y = f a}"
    by (rule chain_fun[OF 1])
  show "P h h2 r"
    using chain_fun[OF 1] flat_lub_in_chain[OF chain_fun[OF 1]] 2 4 unfolding execute_def fun_lub_def
    by force
qed

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 (unfold comp_def, rule ccpo.admissibleI, clarify)
  fix A :: "('heap ⇒ 'e + 'result × 'heap) set"
  let ?lub = "Prog (fun_lub (flat_lub (Inl e)) A)"
  fix h h' h2 h2' r r'
  assume 1: "Complete_Partial_Order.chain (fun_ord (flat_ord (Inl e))) A"
    and 2 [rule_format]: "∀xa∈A. ∀h h' h2 h2' r r'. h ⊢ Prog xa = Inr (r, h2)
                          ⟶ h' ⊢ Prog xa = Inr (r', h2') ⟶ P h h' h2 h2' r r'"
    and 4: "h ⊢ Prog (fun_lub (flat_lub (Inl e)) A) = Inr (r, h2)"
    and 5: "h' ⊢ Prog (fun_lub (flat_lub (Inl e)) A) = Inr (r', h2')"
  have h1:"⋀a. Complete_Partial_Order.chain (flat_ord (Inl e)) {y. ∃f∈A. y = f a}"
    by (rule chain_fun[OF 1])
  have "h ⊢ ?lub ∈ {y. ∃f∈A. y = f h}"
    using flat_lub_in_chain[OF h1] 4
    unfolding execute_def fun_lub_def
    by auto
  moreover have "h' ⊢ ?lub ∈ {y. ∃f∈A. y = f h'}"
    using flat_lub_in_chain[OF h1] 5
    unfolding execute_def fun_lub_def
    by auto
  ultimately obtain f where
    "f ∈ A" and
    "h ⊢ Prog f = Inr (r, h2)" and
    "h' ⊢ Prog f = Inr (r', h2')"
    using 1 4 5
    apply(auto simp add:  chain_def fun_ord_def flat_ord_def execute_def)[1]
    by (metis Inl_Inr_False)
  then show "P h h' h2 h2' r r'"
    by(fact 2)
qed

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"
  by(simp add: dom_prog_lub_def img_lub_def fun_lub_def flat_lub_def error_def)

lemma dom_prog_interpretation: "partial_function_definitions dom_prog_ord dom_prog_lub"
proof -
  have "partial_function_definitions (fun_ord (flat_ord (Inl NonTerminationException)))
                                     (fun_lub (flat_lub (Inl NonTerminationException)))"
```

```
    by (rule partial_function_lift) (rule flat_interpretation)
  then show ?thesis
    apply (simp add: dom_prog_lub_def dom_prog_ord_def flat_interpretation execute_def)
    using partial_function_image prog.expand prog.sel by blast
qed

interpretation dom_prog: partial_function_definitions dom_prog_ord dom_prog_lub
  rewrites "dom_prog_lub {} ≡ error NonTerminationException"
  by (fact dom_prog_interpretation)(simp add: dom_prog_lub_empty)

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 (rule admissible_fun[OF dom_prog_interpretation])
  fix x
  show "ccpo.admissible dom_prog_lub dom_prog_ord (λa. ∀h h' r. h ⊢ a →r r ⟶ h ⊢ a →h h'
        ⟶ P x h h' r)"
    unfolding dom_prog_ord_def dom_prog_lub_def
  proof (intro admissible_image partial_function_lift flat_interpretation)
    show "ccpo.admissible (fun_lub (flat_lub (Inl NonTerminationException)))
                          (fun_ord (flat_ord (Inl NonTerminationException)))
     ((λa. ∀h h' r. h ⊢ a →r r ⟶ h ⊢ a →h h' ⟶ P x h h' r) ∘ Prog)"
      by(auto simp add: execute_admissible returns_result_def returns_heap_def split: sum.splits)
  next
    show "⋀x y. (λb. b ⊢ x) = (λb. b ⊢ y) ⟹ x = y"
      by(simp add: execute_def prog.expand)
  next
    show "⋀x. (λb. b ⊢ Prog x) = x"
      by(simp add: execute_def)
  qed
qed

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 (rule admissible_fun[OF dom_prog_interpretation])
  fix x
  show "ccpo.admissible dom_prog_lub dom_prog_ord (λa. ∀h h2 h' h2' r r2. h ⊢ a →r r
                  ⟶ h ⊢ a →h h' ⟶ h2 ⊢ a →r r2 ⟶ h2 ⊢ a →h h2' ⟶ P x h h2 h' h2' r r2)"
    unfolding dom_prog_ord_def dom_prog_lub_def
  proof (intro admissible_image partial_function_lift flat_interpretation)
    show "ccpo.admissible (fun_lub (flat_lub (Inl NonTerminationException)))
                          (fun_ord (flat_ord (Inl NonTerminationException)))
     ((λa. ∀h h2 h' h2' r r2. h ⊢ a →r r ⟶ h ⊢ a →h h' ⟶ h2 ⊢ a →r r2 ⟶ h2 ⊢ a →h h2'
                ⟶ P x h h2 h' h2' r r2) ∘ Prog)"
      by(auto simp add: returns_result_def returns_heap_def intro!: ccpo.admissibleI
          dest!: ccpo.admissibleD[OF execute_admissible2[where P="P x"]]
          split: sum.splits)
  next
    show "⋀x y. (λb. b ⊢ x) = (λb. b ⊢ y) ⟹ x = y"
      by(simp add: execute_def prog.expand)
  next
    show "⋀x. (λb. b ⊢ Prog x) = x"
      by(simp add: execute_def)
  qed
qed

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"
  using step defined dom_prog.fixp_induct_uc[of U F C, OF mono eq inverse2 admissible_dom_prog, of P]
  by (metis assms(6) error_returns_heap)

declaration ‹Partial_Function.init "dom_prog" @{term dom_prog.fixp_fun}
  @{term dom_prog.mono_body} @{thm dom_prog.fixp_rule_uc} @{thm dom_prog.fixp_induct_uc}
  (SOME @{thm fixp_induct_dom_prog})›


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(auto simp add: dom_prog_ord_def img_ord_def fun_ord_def flat_ord_def)[1]
  fix x
  assume "x ⊢ f ≠ x ⊢ g"
  then show "x ⊢ f = Inl NonTerminationException"
    using assms[where h=x]
    by(auto simp add: returns_error_def split: sum.splits)
qed

lemma dom_prog_ordE:
  assumes "dom_prog_ord x y"
  obtains "h ⊢ x →e NonTerminationException" | " h ⊢ x = h ⊢ y"
  using assms unfolding dom_prog_ord_def img_ord_def fun_ord_def flat_ord_def
  using returns_error_def by force


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 (rule monotoneI)
  fix f g :: "'a ⇒ ('heap, exception, 'result) prog"
  assume fg: "dom_prog.le_fun f g"
  from mf
  have 1: "dom_prog_ord (B f) (B g)" by (rule monotoneD) (rule fg)
  from mg
  have 2: "⋀y'. dom_prog_ord (C y' f) (C y' g)" by (rule monotoneD) (rule fg)

  have "dom_prog_ord (B f ⤜ (λy. C y f)) (B g ⤜ (λy. C y f))"
    (is "dom_prog_ord ?L ?R")
  proof (rule dom_prog_ordI)
    fix h
    from 1 show "h ⊢ ?L →e NonTerminationException ∨ h ⊢ ?L = h ⊢ ?R"
      apply(rule dom_prog_ordE)
       apply(auto)[1]
      using bind_cong by fastforce
  qed
  also
  have h1: "dom_prog_ord (B g ⤜ (λy'. C y' f)) (B g ⤜ (λy'. C y' g))"
    (is "dom_prog_ord ?L ?R")
  proof (rule dom_prog_ordI)

    fix h
    show "h ⊢ ?L →e NonTerminationException ∨ h ⊢ ?L = h ⊢ ?R"
    proof (cases "h ⊢ ok (B g)")
      case True
```

```
      then obtain x h' where x: "h ⊢ B g →r x" and h': "h ⊢ B g →h h'"
        by blast
      then have "dom_prog_ord (C x f) (C x g)"
        using 2 by simp
      then show ?thesis
        using x h'
        apply(auto intro!: bind_returns_error_I3 dest: returns_result_eq dest!: dom_prog_ordE)[1]
        apply(auto simp add: execute_bind_simp)[1]
        using "2" dom_prog_ordE by metis
    next
      case False
      then obtain e where e: "h ⊢ B g →e e"
        by(simp add: is_OK_def returns_error_def split: sum.splits)
      have "h ⊢ B g ≫= (λy'. C y' f) →e e"
        using e by(auto)
      moreover have "h ⊢ B g ≫= (λy'. C y' g) →e e"
        using e by auto
      ultimately show ?thesis
        using bind_returns_error_eq by metis
    qed
  qed
  finally (dom_prog.leq_trans)
  show "dom_prog_ord (B f ≫= (λy. C y f)) (B g ≫= (λy'. C y' g))" .
qed

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)"
  using assms
  apply (induct xs)
  by(auto simp add: call_mono dom_prog.const_mono intro!: bind_mono)

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)"
  using assms
  apply (induct xs)
  by(auto simp add: call_mono dom_prog.const_mono intro!: bind_mono)

lemma sorted_list_set_cong [simp]:
  "sorted_list_of_set (fset FS) = sorted_list_of_set (fset FS') ⟷ FS = FS'"
  by auto

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 .
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" .
lemma get_M_is_l_get_M: "l_get_M get_Object type_wf object_ptr_kinds"
  by (simp add: a_type_wf_def get_Object_type_wf l_get_M_def)
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
  apply(unfold_locales)
   apply (simp add: get_Object_type_wf local.type_wf_Object)
  by (simp add: a_type_wf_def get_Object_type_wf)
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
  by (simp add: l_get_M_Object_lemmas_def l_type_wf_Object_axioms)

lemma object_ptr_kinds_M_reads:
  "reads (⋃object_ptr. {preserved (get_M_Object object_ptr RObject.nothing)}) object_ptr_kinds_M h h'"
  apply(auto simp add: object_ptr_kinds_M_defs get_Object_type_wf type_wf_defs reads_def
      preserved_def get_M_defs
      split: option.splits)[1]
  using a_type_wf_def get_Object_type_wf by blast+


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
   apply (simp add: get_M_is_l_get_M l_put_M_def)
  by (simp add: get_M_Object_def)
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
  apply(unfold_locales)
  using get_Object_type_wf l_type_wf_Object.type_wf_Object local.l_type_wf_Object_axioms apply blast
  by (simp add: a_type_wf_def get_Object_type_wf)
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
  by (simp add: l_put_M_Object_lemmas_def l_type_wf_Object_axioms)


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)"
  by(auto simp add: check_in_heap_def)
lemma check_in_heap_pure [simp]: "pure (check_in_heap ptr) h"
  by(auto simp add: check_in_heap_def intro!: bind_pure_I)
lemma check_in_heap_is_OK [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ ok (check_in_heap ptr ⤜ f) = h ⊢ ok (f ())"
  by(simp add: check_in_heap_def)
lemma check_in_heap_returns_result [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ (check_in_heap ptr ⤜ f) →_r x = h ⊢ f () →_r x"
  by(simp add: check_in_heap_def)
lemma check_in_heap_returns_heap [simp]:
  "ptr |∈| object_ptr_kinds h ⟹ h ⊢ (check_in_heap ptr ⤜ f) →_h h' = h ⊢ f () →_h h'"
  by(simp add: check_in_heap_def)

lemma check_in_heap_reads:
  "reads {preserved (get_M object_ptr nothing)} (check_in_heap object_ptr) h h'"
  apply(simp add: check_in_heap_def reads_def preserved_def)
  by (metis a_type_wf_def get_M_Object_ok get_M_Object_ptr_in_heap is_OK_returns_result_E
      is_OK_returns_result_I unit_all_impI)
```

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})))"
  by(simp add: invoke_def)

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}))))))"
  by(simp add: invoke_def)
lemmas invoke_splits = invoke_split invoke_split_asm

lemma invoke_ptr_in_heap: "h ⊢ ok (invoke xs ptr args) ⟹ ptr |∈| object_ptr_kinds h"
  by (metis bind_is_OK_E check_in_heap_ptr_in_heap invoke_def is_OK_returns_heap_I)

lemma invoke_pure [simp]: "pure (invoke [] ptr args) h"
  by(auto simp add: invoke_def intro!: bind_pure_I)

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)"
  by(simp add: invoke_def)
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"
  by(simp add: invoke_def)
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'"
```

by(*simp add: invoke_def*)

lemma *invoke_not [simp]: "¬Pred ptr ⟹ invoke ((Pred, f) # xs) ptr args = invoke xs ptr args"*
by(*auto simp add: invoke_def*)

lemma *invoke_empty [simp]: "¬h ⊢ ok (invoke [] ptr args)"*
by(*auto simp add: invoke_def check_in_heap_def*)

lemma *invoke_empty_reads [simp]: "∀P ∈ S. reflp P ∧ transp P ⟹ reads S (invoke [] ptr args) h h'"*
apply(*simp add: invoke_def reads_def preserved_def*)
by (*meson bind_returns_result_E error_returns_result*)

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)"*
by(*auto simp add: get*$_{Object}$*_def put*$_{Object}$*_def split: option.splits Option.bind_splits*)

lemma *object_ptr_kinds_simp [simp]: "object_ptr_kinds (put*$_{Object}$ *ptr obj h) = object_ptr_kinds h |∪| {|ptr|}"*
by(*auto simp add: object_ptr_kinds_def put*$_{Object}$*_def split: option.splits*)

lemma *type_wf_put_I:*
assumes *"type_wf h"*
shows *"type_wf (put*$_{Object}$ *ptr obj h)"*
using *assms*
by(*auto simp add: type_wf_defs split: option.splits*)

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"*
using *assms*
by(*auto simp add: type_wf_defs split: option.splits if_splits*)

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"*
using *assms*
by(*auto simp add: type_wf_defs split: option.splits if_splits*)

5.2.3 Preserving Types

lemma *type_wf_preserved: "type_wf h = type_wf h'"*
by(*auto simp add: type_wf_defs*)

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'"*
using *assms*
apply(*auto simp add: object_ptr_kinds_def preserved_def get_M_defs get*$_{Object}$*_def*
split: option.splits)[1]
apply *(metis (mono_tags, lifting) domIff error_returns_result fmdom.rep_eq fmember.rep_eq*
old.unit.exhaust option.case_eq_if return_returns_result)
by *(metis (mono_tags, lifting) domIff error_returns_result fmdom.rep_eq fmember.rep_eq*
old.unit.exhaust option.case_eq_if return_returns_result)

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 -
  {
    fix object_ptr w
    have "preserved (get_M_Object object_ptr RObject.nothing) h h'"
      apply(rule writes_small_big[OF assms])
      by auto
  }
  then show ?thesis
    using object_ptr_kinds_preserved_small by blast
qed

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 .
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"
  using assms
  by(auto simp add: node_ptr_kinds_M_defs object_ptr_kinds_M_defs node_ptr_kinds_def)


global_interpretation l_dummy defines get_M_Node = "l_get_M.a_get_M get_Node" .
lemma get_M_is_l_get_M: "l_get_M get_Node type_wf node_ptr_kinds"
  apply(simp add: get_Node_type_wf l_get_M_def)
  by (metis ObjectClass.a_type_wf_def ObjectClass.get_Object_type_wf bind_eq_None_conv get_Node_def
      node_ptr_kinds_commutes option.simps(3))
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 by unfold_locales

interpretation l_get_M get_Node type_wf node_ptr_kinds
  apply(unfold_locales)
   apply (simp add: get_Node_type_wf local.type_wf_Node)
  by (meson NodeMonad.get_M_is_l_get_M l_get_M_def)
lemmas get_M_Node_ok = get_M_ok[folded get_M_Node_def]
end

global_interpretation l_get_M_Node_lemmas type_wf by unfold_locales

lemma node_ptr_kinds_M_reads:
  "reads (⋃object_ptr. {preserved (get_M_Object object_ptr RObject.nothing)}) node_ptr_kinds_M h h'"
```

```
  using object_ptr_kinds_M_reads
  apply(simp add: reads_def node_ptr_kinds_M_defs node_ptr_kinds_def
      object_ptr_kinds_M_reads preserved_def)
  by (metis (mono_tags, hide_lams)  object_ptr_kinds_preserved_small old.unit.exhaust preserved_def)

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
   apply (simp add: get_M_is_l_get_M l_put_M_def)
  by (simp add: get_M_Node_def)

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 by unfold_locales

interpretation l_put_M type_wf node_ptr_kinds get_Node put_Node
  apply(unfold_locales)
   apply (simp add: get_Node_type_wf local.type_wf_Node)
  by (meson NodeMonad.get_M_is_l_get_M l_get_M_def)
lemmas put_M_Node_ok = put_M_ok[folded put_M_Node_def]
end

global_interpretation l_put_M_Node_lemmas type_wf by unfold_locales

lemma get_M_Object_preserved1 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x)) ⟹ h ⊢ put_M_Node node_ptr setter v →_h h'

    ⟹ preserved (get_M_Object object_ptr getter) h h'"
  apply(cases "cast node_ptr = object_ptr")
  by(auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs get_Node_def preserved_def put_Node_def

      bind_eq_Some_conv
      split: option.splits)

lemma get_M_Object_preserved2 [simp]:
  "cast node_ptr ≠ object_ptr ⟹ h ⊢ put_M_Node node_ptr setter v →_h h'
    ⟹ preserved (get_M_Object object_ptr getter) h h'"
  by(auto simp add: put_M_defs get_M_defs get_Node_def put_Node_def ObjectMonad.get_M_defs preserved_def

      split: option.splits dest: get_heap_E)
lemma get_M_Object_preserved3 [simp]:
  "h ⊢ put_M_Node node_ptr setter v →_h h' ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))

    ⟹ preserved (get_M_Object object_ptr getter) h h'"
  apply(cases "cast node_ptr ≠ object_ptr")
  by(auto simp add: put_M_defs get_M_defs get_Node_def put_Node_def ObjectMonad.get_M_defs preserved_def

      split: option.splits bind_splits dest: get_heap_E)

lemma get_M_Object_preserved4 [simp]:
  "cast node_ptr ≠ object_ptr ⟹ h ⊢ put_M_Object object_ptr setter v →_h h'
      ⟹ preserved (get_M_Node node_ptr getter) h h'"
  by(auto simp add: ObjectMonad.put_M_defs get_M_defs get_Node_def ObjectMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
```

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)"
```

```
  by(auto simp add: get_Node_def)

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 {||})"
  by(auto simp add: node_ptr_kinds_def)

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)"
  using assms
  apply(auto simp add: type_wf_defs split: option.splits)[1]
  using cast_Object2Node_none is_node_kind_def apply blast
  using cast_Object2Node_none is_node_kind_def apply blast
  done

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"
  using assms
  by(auto simp add: type_wf_defs elim!: ObjectMonad.type_wf_put_ptr_not_in_heap_E
      split: option.splits if_splits)

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"
  using assms
  apply(auto simp add: type_wf_defs split: option.splits if_splits)
  by (metis ObjectClass.get_Object_type_wf bind.bind_lunit get_Node_def is_node_kind_def option.collapse)
```

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'"
  by(simp add: node_ptr_kinds_def preserved_def object_ptr_kinds_preserved_small[OF assms])

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'"
  using writes_small_big[OF assms]
  apply(simp add: reflp_def transp_def preserved_def node_ptr_kinds_def)
  by (metis assms object_ptr_kinds_preserved)


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'"
  using type_wf_preserved allI[OF assms(2), of id, simplified]
  apply(auto simp add: type_wf_defs)
   apply(auto simp add: preserved_def get_M_defs node_ptr_kinds_small[OF assms(1)]
      split: option.splits, force)[1]
  by(auto simp add: preserved_def get_M_defs node_ptr_kinds_small[OF assms(1)]
```

```
      split: option.splits, force)[1]

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 -
  have "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"
    using assms type_wf_preserved_small by fast
  with assms(1) assms(2) show ?thesis
    apply(rule writes_small_big)
    by(auto simp add: reflp_def transp_def)
qed
end
```

5.4 Element (ElementMonad)

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

```
theory ElementMonad
  imports
    NodeMonad
    "../classes/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 .
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"
  using assms
  by(auto simp add: element_ptr_kinds_M_defs node_ptr_kinds_M_defs element_ptr_kinds_def)

lemma element_ptr_kinds_M_reads:
  "reads (⋃element_ptr. {preserved (get_M_Object element_ptr RObject.nothing)}) element_ptr_kinds_M h h'"
  apply(simp add: reads_def node_ptr_kinds_M_defs element_ptr_kinds_M_defs element_ptr_kinds_def
      node_ptr_kinds_M_reads preserved_def)
  by (metis (mono_tags, hide_lams) node_ptr_kinds_small old.unit.exhaust preserved_def)

global_interpretation l_dummy defines get_M_Element = "l_get_M.a_get_M get_Element" .
lemma get_M_is_l_get_M: "l_get_M get_Element type_wf element_ptr_kinds"
  apply(simp add: get_Element_type_wf l_get_M_def)
  by (metis (no_types, lifting) ObjectClass.get_Object_type_wf ObjectClass.type_wf_defs
      bind_eq_Some_conv bind_eq_Some_conv element_ptr_kinds_commutes get_Element_def
      get_Node_def get_Object_def node_ptr_kinds_commutes option.simps(3))
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 by unfold_locales

interpretation l_get_M get_Element type_wf element_ptr_kinds
  apply(unfold_locales)
   apply (simp add: get_Element_type_wf local.type_wf_Element)
  by (meson ElementMonad.get_M_is_l_get_M l_get_M_def)
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 by unfold_locales


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
   apply (simp add: get_M_is_l_get_M l_put_M_def)
  by (simp add: get_M_Element_def)

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 by unfold_locales

interpretation l_put_M type_wf element_ptr_kinds get_Element put_Element
  apply(unfold_locales)
   apply (simp add: get_Element_type_wf local.type_wf_Element)
  by (meson ElementMonad.get_M_is_l_get_M l_get_M_def)

lemmas put_M_Element_ok = put_M_ok[folded put_M_Element_def]
end

global_interpretation l_put_M_Element_lemmas type_wf by unfold_locales


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"
  by(auto simp add: put_M_defs get_M_defs split: option.splits)
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'"
  by(auto simp add: put_M_defs get_M_defs preserved_def split: option.splits dest: get_heap_E)
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'"
  apply(cases "element_ptr = element_ptr'")
  by(auto simp add: put_M_defs get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  apply(cases "cast element_ptr = object_ptr")
  by (auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs NodeMonad.get_M_defs get_Element_def
      get_Node_def preserved_def put_Element_def put_Node_def bind_eq_Some_conv
      split: option.splits)
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'"
```

```
  apply(cases "cast element_ptr = node_ptr")
  by(auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs NodeMonad.get_M_defs get_Element_def
      get_Node_def preserved_def put_Element_def put_Node_def bind_eq_Some_conv
      split: option.splits)

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'"
  by(auto simp add: put_M_defs get_M_defs get_Element_def put_Element_def NodeMonad.get_M_defs preserved_def

      split: option.splits dest: get_heap_E)
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'"
  apply(cases "cast element_ptr ≠ node_ptr")
  by(auto simp add: put_M_defs get_M_defs get_Element_def put_Element_def NodeMonad.get_M_defs preserved_def

      split: option.splits bind_splits dest: get_heap_E)

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'"
  by(auto simp add: NodeMonad.put_M_defs get_M_defs get_Element_def NodeMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)

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'"
  by(auto simp add: put_M_defs get_M_defs get_Element_def get_Node_def put_Node_def put_Element_def
      ObjectMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  apply(cases "cast element_ptr ≠ object_ptr")
  by(auto simp add: put_M_defs get_M_defs get_Element_def put_Element_def get_Node_def put_Node_def
      ObjectMonad.get_M_defs preserved_def
      split: option.splits bind_splits dest: get_heap_E)

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'"
  by(auto simp add: ObjectMonad.put_M_defs get_M_defs get_Element_def get_Node_def put_Node_def
      ObjectMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
```

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"
  by(auto simp add: new_element_def split: prod.splits)
```

lemma *new_element_ptr_in_heap:*
assumes *"h ⊢ new_element →_h h'"*
and *"h ⊢ new_element →_r new_element_ptr"*
shows *"new_element_ptr |∈| element_ptr_kinds h'"*
using *assms*
unfolding *new_element_def*
by *(auto simp add: new_element_def new_Element_def Let_def put_Element_ptr_in_heap is_OK_returns_result_I elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_ptr_not_in_heap:*
assumes *"h ⊢ new_element →_h h'"*
and *"h ⊢ new_element →_r new_element_ptr"*
shows *"new_element_ptr |∉| element_ptr_kinds h"*
using *assms new_Element_ptr_not_in_heap*
by *(auto simp add: new_element_def split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_new_ptr:*
assumes *"h ⊢ new_element →_h h'"*
and *"h ⊢ new_element →_r new_element_ptr"*
shows *"object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_element_ptr|}"*
using *assms new_Element_new_ptr*
by *(auto simp add: new_element_def split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_is_element_ptr:*
assumes *"h ⊢ new_element →_r new_element_ptr"*
shows *"is_element_ptr new_element_ptr"*
using *assms new_Element_is_element_ptr*
by *(auto simp add: new_element_def elim!: bind_returns_result_E split: prod.splits)*

lemma *new_element_child_nodes:*
assumes *"h ⊢ new_element →_h h'"*
assumes *"h ⊢ new_element →_r new_element_ptr"*
shows *"h' ⊢ get_M new_element_ptr child_nodes →_r []"*
using *assms*
by *(auto simp add: get_M_defs new_element_def new_Element_def Let_def split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_tag_type:*
assumes *"h ⊢ new_element →_h h'"*
assumes *"h ⊢ new_element →_r new_element_ptr"*
shows *"h' ⊢ get_M new_element_ptr tag_type →_r ''''"*
using *assms*
by *(auto simp add: get_M_defs new_element_def new_Element_def Let_def split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_attrs:*
assumes *"h ⊢ new_element →_h h'"*
assumes *"h ⊢ new_element →_r new_element_ptr"*
shows *"h' ⊢ get_M new_element_ptr attrs →_r fmempty"*
using *assms*
by *(auto simp add: get_M_defs new_element_def new_Element_def Let_def split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_shadow_root_opt:*
assumes *"h ⊢ new_element →_h h'"*
assumes *"h ⊢ new_element →_r new_element_ptr"*
shows *"h' ⊢ get_M new_element_ptr shadow_root_opt →_r None"*
using *assms*
by *(auto simp add: get_M_defs new_element_def new_Element_def Let_def split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)*

lemma *new_element_get_M$_{Object}$:*
"h ⊢ new_element →$_h$ h' ⟹ h ⊢ new_element →$_r$ new_element_ptr ⟹ ptr ≠ cast new_element_ptr
⟹ preserved (get_M$_{Object}$ ptr getter) h h'"
by *(auto simp add: new_element_def ObjectMonad.get_M_defs preserved_def*
split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma *new_element_get_M$_{Node}$:*
"h ⊢ new_element →$_h$ h' ⟹ h ⊢ new_element →$_r$ new_element_ptr ⟹ ptr ≠ cast new_element_ptr
⟹ preserved (get_M$_{Node}$ ptr getter) h h'"
by *(auto simp add: new_element_def NodeMonad.get_M_defs preserved_def*
split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma *new_element_get_M$_{Element}$:*
"h ⊢ new_element →$_h$ h' ⟹ h ⊢ new_element →$_r$ new_element_ptr ⟹ ptr ≠ new_element_ptr
⟹ preserved (get_M$_{Element}$ ptr getter) h h'"
by *(auto simp add: new_element_def get_M_defs preserved_def*
split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)

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)"
by *(auto simp add: get$_{Element}$_def split: option.splits Option.bind_splits)*

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 {||})"
by *(auto simp add: element_ptr_kinds_def is_node_ptr_kind_def split: option.splits)*

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)"*
using *assms*
by *(auto simp add: type_wf_defs split: option.splits)*

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"*
using *assms*
by *(auto simp add: type_wf_defs elim!: NodeMonad.type_wf_put_ptr_not_in_heap_E*
split: option.splits if_splits)

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"*
using *assms*
apply *(auto simp add: type_wf_defs split: option.splits if_splits)[1]*
apply *(case_tac "x2 = cast element_ptr")*
apply *(drule_tac x=element_ptr* **in** *allE)*
apply *(auto)[1]*
apply *(metis (no_types, lifting) NodeClass.get$_{Object}$_type_wf assms(2) bind.bind_lunit*
cast$_{Node2Element}$_inv cast$_{Object2Node}$_inv get$_{Element}$_def get$_{Node}$_def option.exhaust_sel)
by *(auto)*

5.4.3 Preserving Types

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

```
  apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs new_Element_def
      new_element_def Let_def put_Element_def put_Node_def put_Object_def get_Element_def
      get_Node_def get_Object_def
      split: prod.splits if_splits elim!: bind_returns_heap_E)[1]
     apply (metis element_ptr_kinds_commutes element_ptrs_def fempty_iff ffmember_filter
      is_element_ptr_ref)
  using element_ptrs_def apply fastforce
   apply (metis (mono_tags, hide_lams) Suc_n_not_le_n element_ptr.sel(1) element_ptr_kinds_commutes
      element_ptrs_def fMax_ge ffmember_filter fimageI is_element_ptr_ref)
  by (metis (no_types, lifting) fMax_finsert fempty_iff fimage_is_fempty max_0L new_Element_def
      new_Element_ptr_not_in_heap)

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"
  using l_new_element.intro new_element_type_wf_preserved
  by blast

lemma put_M_Element_tag_type_type_wf_preserved [simp]:
  "h ⊢ put_M element_ptr tag_type_update v →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs
      Let_def put_M_defs get_M_defs put_Element_def put_Node_def put_Object_def
      get_Element_def get_Node_def get_Object_def
      split: prod.splits option.splits Option.bind_splits elim!: bind_returns_heap_E)[1]
      apply (metis option.distinct(1))
     apply (metis bind.bind_lunit  cast_Object2Node_none)
    apply (metis option.distinct(1))
   apply (metis bind.bind_lunit  cast_Object2Node_none)
  by (metis bind.bind_lunit cast_Node2Element_none cast_Object2Node_inv)


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'"
  apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs
      Let_def put_M_defs get_M_defs put_Element_def put_Node_def put_Object_def
      get_Element_def get_Node_def get_Object_def
      split: prod.splits option.splits Option.bind_splits elim!: bind_returns_heap_E)[1]
      apply (metis option.distinct(1))
     apply (metis bind.bind_lunit  cast_Object2Node_none)
    apply (metis option.distinct(1))
   apply (metis bind.bind_lunit  cast_Object2Node_none)
  by (metis bind.bind_lunit cast_Node2Element_none cast_Object2Node_inv)

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'"
  apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs  Let_def
      put_M_defs get_M_defs put_Element_def put_Node_def put_Object_def get_Element_def
      get_Node_def get_Object_def
      split: prod.splits option.splits Option.bind_splits elim!: bind_returns_heap_E)[1]
      apply (metis option.distinct(1))
     apply (metis bind.bind_lunit  cast_Object2Node_none)
    apply (metis option.distinct(1))
   apply (metis bind.bind_lunit  cast_Object2Node_none)
  by (metis bind.bind_lunit cast_Node2Element_none cast_Object2Node_inv)

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'"
  apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs
      Let_def put_M_defs get_M_defs put_Element_def put_Node_def put_Object_def
      get_Element_def get_Node_def get_Object_def
      split: prod.splits option.splits Option.bind_splits elim!: bind_returns_heap_E)[1]
      apply (metis option.distinct(1))
```

```
    apply (metis bind.bind_lunit  cast_Object2Node_none)
   apply (metis option.distinct(1))
  apply (metis bind.bind_lunit  cast_Object2Node_none)
 by (metis bind.bind_lunit cast_Node2Element_none cast_Object2Node_inv)

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'"
 using assms
 apply(auto simp add: put_M_defs put_Element_def put_Node_def put_Object_def
     elim!: bind_returns_heap_E2 dest!: get_heap_E)[1]
 by (meson get_M_Element_ptr_in_heap is_OK_returns_result_I)

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'"
 using writes_small_big[OF assms]
 apply(simp add: reflp_def transp_def preserved_def element_ptr_kinds_def)
 by (metis assms node_ptr_kinds_preserved)


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'"
 by(simp add: element_ptr_kinds_def node_ptr_kinds_def preserved_def
     object_ptr_kinds_preserved_small[OF assms])

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'"
 using type_wf_preserved_small[OF assms(1) assms(2)] allI[OF assms(3), of id, simplified]
 apply(auto simp add: type_wf_defs )[1]
  apply(auto simp add: preserved_def get_M_defs element_ptr_kinds_small[OF assms(1)]
     split: option.splits,force)[1]
 by(auto simp add: preserved_def get_M_defs element_ptr_kinds_small[OF assms(1)]
     split: option.splits,force)

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 -
 have "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"
   using assms type_wf_preserved_small by fast
 with assms(1) assms(2) show ?thesis
   apply(rule writes_small_big)
   by(auto simp add: reflp_def transp_def)
qed

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
 apply(auto simp add: type_wf_defs NodeClass.type_wf_defs ObjectClass.type_wf_defs
     node_ptr_kinds_def object_ptr_kinds_def is_node_ptr_kind_def
```

```
    get_Element_def get_Node_def get_Object_def)[1]
   apply (metis (mono_tags, lifting) comp_apply ffmember_filter fimage_eqI
    is_node_ptr_kind_cast node_ptr_casts_commute2 option.sel)
  apply (metis (no_types, lifting) comp_apply element_ptr_kinds_commutes ffmember_filter
    fmdom_filter fmfilter_alt_defs(1) heap.sel node_ptr_kinds_commutes object_ptr_kinds_def)
 by (metis comp_eq_dest_lhs element_ptr_kinds_commutes fmdom_notI fmdrop_lookup heap.sel
    node_ptr_kinds_commutes object_ptr_kinds_def)

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 .
lemmas character_data_ptr_kinds_M_defs = a_ptr_kinds_M_def

lemma character_data_ptr_kinds_M_eq:
  assumes "|h ⊢ node_ptr_kinds_M|_r = |h' ⊢ node_ptr_kinds_M|_r"
  shows "|h ⊢ character_data_ptr_kinds_M|_r = |h' ⊢ character_data_ptr_kinds_M|_r"
  using assms
  by(auto simp add: character_data_ptr_kinds_M_defs node_ptr_kinds_M_defs
      character_data_ptr_kinds_def)

lemma character_data_ptr_kinds_M_reads:
  "reads (⋃node_ptr. {preserved (get_M_Object node_ptr RObject.nothing)}) character_data_ptr_kinds_M h h'"
  using node_ptr_kinds_M_reads
  apply(simp add: reads_def node_ptr_kinds_M_defs character_data_ptr_kinds_M_defs
      character_data_ptr_kinds_def preserved_def)
  by (metis (mono_tags, hide_lams) node_ptr_kinds_small old.unit.exhaust preserved_def)

global_interpretation l_dummy defines get_M_CharacterData = "l_get_M.a_get_M get_CharacterData" .
lemma get_M_is_l_get_M: "l_get_M get_CharacterData type_wf character_data_ptr_kinds"
  apply(simp add: get_CharacterData_type_wf l_get_M_def)
  by (metis (no_types, hide_lams) NodeMonad.get_M_is_l_get_M bind_eq_Some_conv
      character_data_ptr_kinds_commutes get_CharacterData_def l_get_M_def option.distinct(1))
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 get_M_CharacterData

locale l_get_M_CharacterData_lemmas = l_type_wf_CharacterData
begin
sublocale l_get_M_Element_lemmas by unfold_locales

interpretation l_get_M get_CharacterData type_wf character_data_ptr_kinds
  apply(unfold_locales)
   apply (simp add: get_CharacterData_type_wf local.type_wf_CharacterData)
  by (meson CharacterDataMonad.get_M_is_l_get_M l_get_M_def)
```

```
lemmas get_M_CharacterData_ok = get_M_ok[folded get_M_CharacterData_def]
end

global_interpretation l_get_M_CharacterData_lemmas type_wf by unfold_locales


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
   apply (simp add: get_M_is_l_get_M l_put_M_def)
  by (simp add: get_M_CharacterData_def)

lemmas put_M_defs = a_put_M_def
adhoc_overloading put_M put_M_CharacterData


locale l_put_M_CharacterData_lemmas = l_type_wf_CharacterData
begin
sublocale l_put_M_Element_lemmas by unfold_locales

interpretation l_put_M type_wf character_data_ptr_kinds get_CharacterData put_CharacterData
  apply(unfold_locales)
  using get_CharacterData_type_wf l_type_wf_CharacterData.type_wf_CharacterData local.l_type_wf_CharacterData_axioms
   apply blast
  by (meson CharacterDataMonad.get_M_is_l_get_M l_get_M_def)
lemmas put_M_CharacterData_ok = put_M_ok[folded put_M_CharacterData_def]
end

global_interpretation l_put_M_CharacterData_lemmas type_wf by unfold_locales


lemma CharacterData_simp1 [simp]:
  "(⋀x. getter (setter (λ_. v) x) = v) ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
   ⟹ h' ⊢ get_M_CharacterData character_data_ptr getter →_r v"
  by(auto simp add: put_M_defs get_M_defs split: option.splits)
lemma CharacterData_simp2 [simp]:
  "character_data_ptr ≠ character_data_ptr'
   ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_CharacterData character_data_ptr' getter) h h'"
  by(auto simp add: put_M_defs get_M_defs preserved_def split: option.splits dest: get_heap_E)
lemma CharacterData_simp3 [simp]: "
  (⋀x. getter (setter (λ_. v) x) = getter x)
  ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_CharacterData character_data_ptr' getter) h h'"
  apply(cases "character_data_ptr = character_data_ptr'")
  by(auto simp add: put_M_defs get_M_defs preserved_def split: option.splits dest: get_heap_E)
lemma CharacterData_simp4 [simp]:
  "h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_Element element_ptr getter) h h'"
  by(auto simp add: put_M_defs ElementMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
lemma CharacterData_simp5 [simp]:
  "h ⊢ put_M_Element element_ptr setter v →_h h'
  ⟹ preserved (get_M_CharacterData character_data_ptr getter) h h'"
  by(auto simp add: ElementMonad.put_M_defs get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
lemma CharacterData_simp6 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_Object object_ptr getter) h h'"
  apply (cases "cast character_data_ptr = object_ptr")
  by(auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs NodeMonad.get_M_defs
      get_CharacterData_def get_Node_def preserved_def put_CharacterData_def put_Node_def
```

```
    bind_eq_Some_conv split: option.splits)
lemma CharacterData_simp7 [simp]:
  "(⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
  ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_Node node_ptr getter) h h'"
  apply(cases "cast character_data_ptr = node_ptr")
  by(auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs NodeMonad.get_M_defs
      get_CharacterData_def get_Node_def preserved_def put_CharacterData_def put_Node_def
      bind_eq_Some_conv split: option.splits)

lemma CharacterData_simp8 [simp]:
  "cast character_data_ptr ≠ node_ptr
   ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
   ⟹ preserved (get_M_Node node_ptr getter) h h'"
  by(auto simp add: put_M_defs get_M_defs get_CharacterData_def put_CharacterData_def NodeMonad.get_M_defs

      preserved_def split: option.splits dest: get_heap_E)
lemma CharacterData_simp9 [simp]:
  "h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
   ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M_Node node_ptr getter) h h'"
  apply(cases "cast character_data_ptr ≠ node_ptr")
  by(auto simp add: put_M_defs get_M_defs get_CharacterData_def put_CharacterData_def
      NodeMonad.get_M_defs preserved_def split: option.splits bind_splits
      dest: get_heap_E)
lemma CharacterData_simp10 [simp]:
  "cast character_data_ptr ≠ node_ptr
   ⟹ h ⊢ put_M_Node node_ptr setter v →_h h'
   ⟹ preserved (get_M_CharacterData character_data_ptr getter) h h'"
  by(auto simp add: NodeMonad.put_M_defs get_M_defs get_CharacterData_def NodeMonad.get_M_defs
      preserved_def split: option.splits dest: get_heap_E)

lemma CharacterData_simp11 [simp]:
  "cast character_data_ptr ≠ object_ptr
  ⟹ h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
  ⟹ preserved (get_M_Object object_ptr getter) h h'"
  by(auto simp add: put_M_defs get_M_defs get_CharacterData_def get_Node_def put_Node_def put_CharacterData_def

      ObjectMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)

lemma CharacterData_simp12 [simp]:
  "h ⊢ put_M_CharacterData character_data_ptr setter v →_h h'
   ⟹ (⋀x. getter (cast (setter (λ_. v) x)) = getter (cast x))
   ⟹ preserved (get_M_Object object_ptr getter) h h'"
  apply(cases "cast character_data_ptr ≠ object_ptr")
   apply(auto simp add: put_M_defs get_M_defs get_CharacterData_def put_CharacterData_def
      get_Node_def put_Node_def ObjectMonad.get_M_defs preserved_def
      split: option.splits bind_splits dest: get_heap_E)[1]
  by(auto simp add: put_M_defs get_M_defs get_CharacterData_def put_CharacterData_def
      get_Node_def put_Node_def ObjectMonad.get_M_defs preserved_def
      split: option.splits bind_splits dest: get_heap_E)[1]

lemma CharacterData_simp13 [simp]:
  "cast character_data_ptr ≠ object_ptr ⟹ h ⊢ put_M_Object object_ptr setter v →_h h'
    ⟹ preserved (get_M_CharacterData character_data_ptr getter) h h'"
  by(auto simp add: ObjectMonad.put_M_defs get_M_defs get_CharacterData_def get_Node_def put_Node_def
      ObjectMonad.get_M_defs preserved_def split: option.splits dest: get_heap_E)

lemma new_element_get_M_CharacterData:
  "h ⊢ new_element →_h h' ⟹ preserved (get_M_CharacterData ptr getter) h h'"
  by(auto simp add: new_element_def get_M_defs preserved_def split: prod.splits option.splits
      elim!: bind_returns_result_E bind_returns_heap_E)
```

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"
  by(auto simp add: new_character_data_def split: prod.splits)

lemma new_character_data_ptr_in_heap:
  assumes "h ⊢ new_character_data →_h h'"
    and "h ⊢ new_character_data →_r new_character_data_ptr"
  shows "new_character_data_ptr |∈| character_data_ptr_kinds h'"
  using assms
  unfolding new_character_data_def
  by(auto simp add: new_character_data_def new_CharacterData_def Let_def put_CharacterData_ptr_in_heap
      is_OK_returns_result_I
      elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_character_data_ptr_not_in_heap:
  assumes "h ⊢ new_character_data →_h h'"
    and "h ⊢ new_character_data →_r new_character_data_ptr"
  shows "new_character_data_ptr |∉| character_data_ptr_kinds h"
  using assms new_CharacterData_ptr_not_in_heap
  by(auto simp add: new_character_data_def split: prod.splits
      elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_character_data_new_ptr:
  assumes "h ⊢ new_character_data →_h h'"
    and "h ⊢ new_character_data →_r new_character_data_ptr"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
  using assms new_CharacterData_new_ptr
  by(auto simp add: new_character_data_def split: prod.splits
      elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_character_data_is_character_data_ptr:
  assumes "h ⊢ new_character_data →_r new_character_data_ptr"
  shows "is_character_data_ptr new_character_data_ptr"
  using assms new_CharacterData_is_character_data_ptr
  by(auto simp add: new_character_data_def elim!: bind_returns_result_E split: prod.splits)

lemma new_character_data_child_nodes:
  assumes "h ⊢ new_character_data →_h h'"
  assumes "h ⊢ new_character_data →_r new_character_data_ptr"
  shows "h' ⊢ get_M new_character_data_ptr val →_r ''''"
  using assms
  by(auto simp add: get_M_defs new_character_data_def new_CharacterData_def Let_def
      split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_character_data_get_M_Object:
  "h ⊢ new_character_data →_h h' ⟹ h ⊢ new_character_data →_r new_character_data_ptr
    ⟹ ptr ≠ cast new_character_data_ptr ⟹ preserved (get_M_Object ptr getter) h h'"
  by(auto simp add: new_character_data_def ObjectMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_character_data_get_M_Node:
  "h ⊢ new_character_data →_h h' ⟹ h ⊢ new_character_data →_r new_character_data_ptr
    ⟹ ptr ≠ cast new_character_data_ptr ⟹ preserved (get_M_Node ptr getter) h h'"
```

```
  by(auto simp add: new_character_data_def NodeMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_character_data_get_M_Element:
  "h ⊢ new_character_data →_h h' ⟹ h ⊢ new_character_data →_r new_character_data_ptr
    ⟹ preserved (get_M_Element ptr getter) h h'"
  by(auto simp add: new_character_data_def ElementMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_character_data_get_M_CharacterData:
  "h ⊢ new_character_data →_h h' ⟹ h ⊢ new_character_data →_r new_character_data_ptr
    ⟹ ptr ≠ new_character_data_ptr ⟹ preserved (get_M_CharacterData ptr getter) h h'"
  by(auto simp add: new_character_data_def get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
```

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)"
  by(auto simp add: get_CharacterData_def split: option.splits Option.bind_splits)

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 {||})"
  by(auto simp add: character_data_ptr_kinds_def is_node_ptr_kind_def split: option.splits)

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)"
  using assms
  by(auto simp add: type_wf_defs split: option.splits)

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"
  using assms
  by(auto simp add: type_wf_defs elim!: ElementMonad.type_wf_put_ptr_not_in_heap_E
      split: option.splits if_splits)

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"
  using assms
  apply(auto simp add: type_wf_defs split: option.splits if_splits)[1]
  apply(case_tac "x2 = cast character_data_ptr")
   apply(auto)[1]
   apply(drule_tac x=character_data_ptr in allE)
    apply(simp)
   apply (metis (no_types, lifting) ElementClass.get_Object_type_wf assms(2) bind.bind_lunit
      cast_Node2CharacterData_inv cast_Object2Node_inv get_CharacterData_def
      get_Node_def option.exhaust_sel)
  by(blast)
```

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'"
```

```
  using assms
  apply(auto simp add: new_element_def new_Element_def Let_def put_Element_def put_Node_def
      elim!: bind_returns_heap_E type_wf_put_ptr_not_in_heap_E
      intro!: type_wf_put_I split: if_splits)[1]
  using CharacterDataClass.type_wf_Element assms new_element_type_wf_preserved apply blast
  using element_ptrs_def apply fastforce
  using CharacterDataClass.type_wf_Element assms new_element_type_wf_preserved apply blast
  by (metis Suc_n_not_le_n element_ptr.sel(1) element_ptrs_def fMax_ge ffmember_filter
      fimage_eqI is_element_ptr_ref)

lemma new_element_is_l_new_element: "l_new_element type_wf"
  using l_new_element.intro new_element_type_wf_preserved
  by blast

lemma put_M_Element_tag_type_type_wf_preserved [simp]:
  "h ⊢ put_M element_ptr tag_type_update v →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
      dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
      apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs split: option.splits)[1]
     apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs split: option.splits)[1]
    apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs split: option.splits)[1]
   apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs split: option.splits)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs split: option.splits)[1]
  using ObjectMonad.type_wf_put_ptr_in_heap_E ObjectMonad.type_wf_put_ptr_not_in_heap_E apply blast
   apply (metis (mono_tags, lifting) bind_eq_Some_conv get_Element_def option.exhaust_sel)
  by (metis (no_types, lifting) option.exhaust_sel )


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'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
      dest!: get_heap_E  elim!: bind_returns_heap_E2
      intro!: type_wf_put_I ElementMonad.type_wf_put_I
      NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
      apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs
      split: option.splits)[1]
     apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs
      split: option.splits)[1]
    apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs
      split: option.splits)[1]
   apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs
      split: option.splits)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs
      split: option.splits)[1]
  using ObjectMonad.type_wf_put_ptr_in_heap_E ObjectMonad.type_wf_put_ptr_not_in_heap_E apply blast
   apply (metis (mono_tags, lifting) ElementMonad.a_get_M_def bind_eq_Some_conv error_returns_result
      get_Element_def get_heap_returns_result option.exhaust_sel option.simps(4))
  by (metis (no_types, lifting) option.exhaust_sel)

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'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
      dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
      apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
     apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
    apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
   apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs
      ElementMonad.get_M_defs split: option.splits)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs
      ElementMonad.get_M_defs split: option.splits)[1]
  using ObjectMonad.type_wf_put_ptr_in_heap_E ObjectMonad.type_wf_put_ptr_not_in_heap_E apply blast
   apply (metis (mono_tags, lifting) ElementMonad.a_get_M_def bind_eq_Some_conv error_returns_result get_Element_de
get_heap_returns_result option.exhaust_sel option.simps(4))
  by (metis (no_types, lifting) option.exhaust_sel)

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'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
      dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
      apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
     apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
    apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs

      ElementMonad.get_M_defs split: option.splits)[1]
   apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs
      ElementMonad.get_M_defs split: option.splits)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs
      ElementMonad.get_M_defs split: option.splits)[1]
  using ObjectMonad.type_wf_put_ptr_in_heap_E ObjectMonad.type_wf_put_ptr_not_in_heap_E apply blast
   apply (metis (mono_tags, lifting) bind_eq_Some_conv get_Element_def)
  by (metis (no_types, lifting) option.exhaust_sel)


lemma new_character_data_type_wf_preserved [simp]:
  "h ⊢ new_character_data →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: new_character_data_def new_CharacterData_def Let_def put_CharacterData_def put_Node_def

      elim!: bind_returns_heap_E type_wf_put_ptr_not_in_heap_E
      intro!: type_wf_put_I ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I

      split: if_splits)[1]
      apply(simp_all add: type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs is_node_kind_def)
  by (meson new_CharacterData_def new_CharacterData_ptr_not_in_heap)

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"
  using l_new_character_data.intro new_character_data_type_wf_preserved
  by blast

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'"
  apply(auto simp add: CharacterDataMonad.put_M_defs put_CharacterData_def put_Node_def
      CharacterDataClass.type_wf_Node CharacterDataClass.type_wf_Object
      is_node_kind_def
      dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
   apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs CharacterDataMonad.get_M_defs
      split: option.splits)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs CharacterDataMonad.get_M_defs
      ObjectClass.a_type_wf_def
      split: option.splits)[1]
   apply (metis (no_types, lifting) bind_eq_Some_conv get_CharacterData_def)
  by metis

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'"
  by(simp add: character_data_ptr_kinds_def node_ptr_kinds_def preserved_def
      object_ptr_kinds_preserved_small[OF assms])

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'"
  using writes_small_big[OF assms]
  apply(simp add: reflp_def transp_def preserved_def character_data_ptr_kinds_def)
  by (metis assms node_ptr_kinds_preserved)


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'"
  using type_wf_preserved_small[OF assms(1) assms(2) assms(3)]
    allI[OF assms(4), of id, simplified] character_data_ptr_kinds_small[OF assms(1)]
  apply(auto simp add: type_wf_defs preserved_def get_M_defs character_data_ptr_kinds_small[OF assms(1)]

      split: option.splits)[1]
   apply(force)
  by force

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 -
  have "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"
    using assms type_wf_preserved_small by fast
  with assms(1) assms(2) show ?thesis
    apply(rule writes_small_big)
    by(auto simp add: reflp_def transp_def)
qed

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
  apply(auto simp add: type_wf_def ElementMonad.type_wf_drop
      l_type_wf_def_CharacterData.a_type_wf_def)[1]
  using type_wf_drop
  by (metis (no_types, lifting) ElementClass.get_Object_type_wf character_data_ptr_kinds_commutes
      fmlookup_drop get_CharacterData_def get_Node_def get_Object_def heap.sel
      node_ptr_kinds_commutes)

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 .
lemmas document_ptr_kinds_M_defs = a_ptr_kinds_M_def

lemma document_ptr_kinds_M_eq:
  assumes "|h ⊢ object_ptr_kinds_M|_r = |h' ⊢ object_ptr_kinds_M|_r"
  shows "|h ⊢ document_ptr_kinds_M|_r = |h' ⊢ document_ptr_kinds_M|_r"
  using assms
  by(auto simp add: document_ptr_kinds_M_defs object_ptr_kinds_M_defs document_ptr_kinds_def)

lemma document_ptr_kinds_M_reads:
  "reads (⋃object_ptr. {preserved (get_M_Object object_ptr RObject.nothing)}) document_ptr_kinds_M h h'"
  using object_ptr_kinds_M_reads

  apply(simp add: reads_def object_ptr_kinds_M_defs document_ptr_kinds_M_defs
      document_ptr_kinds_def preserved_def)
  by (metis (mono_tags, hide_lams) object_ptr_kinds_preserved_small old.unit.exhaust preserved_def)

global_interpretation l_dummy defines get_M_Document = "l_get_M.a_get_M get_Document" .
lemma get_M_is_l_get_M: "l_get_M get_Document type_wf document_ptr_kinds"
  apply(simp add: get_Document_type_wf l_get_M_def)
  by (metis ObjectClass.get_Object_type_wf ObjectClass.type_wf_defs bind_eq_None_conv
      document_ptr_kinds_commutes get_Document_def option.simps(3))
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 by unfold_locales

interpretation l_get_M get_Document type_wf document_ptr_kinds
  apply(unfold_locales)
   apply (simp add: get_Document_type_wf local.type_wf_Document)
  by (meson DocumentMonad.get_M_is_l_get_M l_get_M_def)
lemmas get_M_Document_ok = get_M_ok[folded get_M_Document_def]
end

global_interpretation l_get_M_Document_lemmas type_wf by unfold_locales


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
   apply (simp add: get_M_is_l_get_M l_put_M_def)
  by (simp add: get_M_Document_def)

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 by unfold_locales

interpretation l_put_M type_wf document_ptr_kinds get_Document put_Document
  apply(unfold_locales)
   apply (simp add: get_Document_type_wf local.type_wf_Document)
  by (meson DocumentMonad.get_M_is_l_get_M l_get_M_def)
lemmas put_M_Document_ok = put_M_ok[folded put_M_Document_def]
end

global_interpretation l_put_M_Document_lemmas type_wf by unfold_locales


lemma document_put_get [simp]:
  "h ⊢ put_M_Document document_ptr setter v →_h h'
     ⟹ (⋀x. getter (setter (λ_. v) x) = v)
     ⟹ h' ⊢ get_M_Document document_ptr getter →_r v"
  by(auto simp add: put_M_defs get_M_defs split: option.splits)
lemma get_M_Mdocument_preserved1 [simp]:
  "document_ptr ≠ document_ptr'
    ⟹ h ⊢ put_M_Document document_ptr setter v →_h h'
    ⟹ preserved (get_M_Document document_ptr' getter) h h'"
  by(auto simp add: put_M_defs get_M_defs preserved_def split: option.splits dest: get_heap_E)
lemma document_put_get_preserved [simp]:
  "h ⊢ put_M_Document document_ptr setter v →_h h'
   ⟹ (⋀x. getter (setter (λ_. v) x) = getter x)
   ⟹ preserved (get_M_Document document_ptr' getter) h h'"
  apply(cases "document_ptr = document_ptr'")
  by(auto simp add: put_M_defs get_M_defs preserved_def split: option.splits dest: get_heap_E)

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'"
  by(auto simp add: put_M_defs get_M_defs NodeMonad.get_M_defs get_Document_def
      put_Document_def get_Node_def preserved_def split: option.splits dest: get_heap_E)

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'"
  by(auto simp add: put_M_defs get_M_defs get_Document_def put_Document_def ObjectMonad.get_M_defs
      preserved_def split: option.splits dest: get_heap_E)
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'"
  apply(cases "cast document_ptr ≠ object_ptr")[1]
  by(auto simp add: put_M_defs get_M_defs get_Document_def put_Document_def
      ObjectMonad.get_M_defs preserved_def
      split: option.splits bind_splits dest: get_heap_E)

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'"
  by(auto simp add: ObjectMonad.put_M_defs get_M_defs get_Document_def ObjectMonad.get_M_defs
      preserved_def split: option.splits dest: get_heap_E)

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'"
  by(auto simp add: put_M_defs ElementMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  by(auto simp add: ElementMonad.put_M_defs get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  by(auto simp add: put_M_defs CharacterDataMonad.get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  by(auto simp add: CharacterDataMonad.put_M_defs get_M_defs preserved_def
      split: option.splits dest: get_heap_E)
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'"
  apply(cases "cast document_ptr = object_ptr")
  by(auto simp add: put_M_defs get_M_defs ObjectMonad.get_M_defs NodeMonad.get_M_defs get_Document_def
      get_Node_def preserved_def put_Document_def put_Node_def bind_eq_Some_conv
      split: option.splits)

lemma new_element_get_M_Document:
  "h ⊢ new_element →_h h' ⟹ preserved (get_M_Document ptr getter) h h'"
  by(auto simp add: new_element_def get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_character_data_get_M_Document:
  "h ⊢ new_character_data →_h h' ⟹ preserved (get_M_Document ptr getter) h h'"
  by(auto simp add: new_character_data_def get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
```

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"
  by(auto simp add: new_document_def split: prod.splits)

lemma new_document_ptr_in_heap:
  assumes "h ⊢ new_document →_h h'"
    and "h ⊢ new_document →_r new_document_ptr"
  shows "new_document_ptr |∈| document_ptr_kinds h'"
  using assms
  unfolding new_document_def
  by(auto simp add: new_document_def new_Document_def Let_def put_Document_ptr_in_heap is_OK_returns_result_I
      elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_document_ptr_not_in_heap:
  assumes "h ⊢ new_document →_h h'"
    and "h ⊢ new_document →_r new_document_ptr"
  shows "new_document_ptr |∉| document_ptr_kinds h"
  using assms new_Document_ptr_not_in_heap
  by(auto simp add: new_document_def split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_document_new_ptr:
  assumes "h ⊢ new_document →_h h'"
    and "h ⊢ new_document →_r new_document_ptr"
  shows "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_document_ptr|}"
  using assms new_Document_new_ptr
  by(auto simp add: new_document_def split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_document_is_document_ptr:
  assumes "h ⊢ new_document →_r new_document_ptr"
  shows "is_document_ptr new_document_ptr"
  using assms new_Document_is_document_ptr
  by(auto simp add: new_document_def elim!: bind_returns_result_E split: prod.splits)

lemma new_document_doctype:
  assumes "h ⊢ new_document →_h h'"
  assumes "h ⊢ new_document →_r new_document_ptr"
  shows "h' ⊢ get_M new_document_ptr doctype →_r ''''"
  using assms
  by(auto simp add: get_M_defs new_document_def new_Document_def Let_def
      split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_document_document_element:
  assumes "h ⊢ new_document →_h h'"
  assumes "h ⊢ new_document →_r new_document_ptr"
  shows "h' ⊢ get_M new_document_ptr document_element →_r None"
  using assms
  by(auto simp add: get_M_defs new_document_def new_Document_def Let_def
      split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)

lemma new_document_disconnected_nodes:
  assumes "h ⊢ new_document →_h h'"
  assumes "h ⊢ new_document →_r new_document_ptr"
  shows "h' ⊢ get_M new_document_ptr disconnected_nodes →_r []"
  using assms
  by(auto simp add: get_M_defs new_document_def new_Document_def Let_def
      split: option.splits prod.splits elim!: bind_returns_result_E bind_returns_heap_E)
```

```
lemma new_document_get_M_Object:
  "h ⊢ new_document →_h h' ⟹ h ⊢ new_document →_r new_document_ptr
    ⟹ ptr ≠ cast new_document_ptr ⟹ preserved (get_M_Object ptr getter) h h'"
  by(auto simp add: new_document_def ObjectMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_document_get_M_Node:
  "h ⊢ new_document →_h h' ⟹ h ⊢ new_document →_r new_document_ptr
    ⟹ preserved (get_M_Node ptr getter) h h'"
  by(auto simp add: new_document_def NodeMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_document_get_M_Element:
  "h ⊢ new_document →_h h' ⟹ h ⊢ new_document →_r new_document_ptr
    ⟹ preserved (get_M_Element ptr getter) h h'"
  by(auto simp add: new_document_def ElementMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
lemma new_document_get_M_CharacterData:
  "h ⊢ new_document →_h h' ⟹ h ⊢ new_document →_r new_document_ptr
    ⟹ preserved (get_M_CharacterData ptr getter) h h'"
  by(auto simp add: new_document_def CharacterDataMonad.get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
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'"
  by(auto simp add: new_document_def get_M_defs preserved_def
      split: prod.splits option.splits elim!: bind_returns_result_E bind_returns_heap_E)
```

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)"
  by(auto simp add: get_Document_def split: option.splits Option.bind_splits)

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 {||})"
  by(auto simp add: document_ptr_kinds_def split: option.splits)

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)"
  using assms
  by(auto simp add: type_wf_defs is_document_kind_def split: option.splits)

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"
  using assms
  by(auto simp add: type_wf_defs elim!: CharacterDataMonad.type_wf_put_ptr_not_in_heap_E
      split: option.splits if_splits)

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"
  using assms
```

apply(*auto simp add: type_wf_defs elim!: CharacterDataMonad.type_wf_put_ptr_in_heap_E*
split: option.splits if_splits)[1]
by *(metis (no_types, lifting) CharacterDataClass.get$_{Object}$_type_wf bind.bind_lunit get$_{Document}$_def*
is_document_kind_def option.collapse)

5.6.3 Preserving Types

lemma *new_element_type_wf_preserved [simp]:*
"h ⊢ new_element →$_h$ h' ⟹ type_wf h = type_wf h'"
apply(*auto simp add: new_element_def new$_{Element}$_def Let_def put$_{Element}$_def put$_{Node}$_def*
DocumentClass.type_wf$_{CharacterData}$ DocumentClass.type_wf$_{Element}$
DocumentClass.type_wf$_{Node}$ DocumentClass.type_wf$_{Object}$
is_node_kind_def element_ptrs_def
elim!: bind_returns_heap_E type_wf_put_ptr_not_in_heap_E
intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I
split: if_splits)[1]
apply *fastforce*
by *(metis Suc_n_not_le_n element_ptr.sel(1) element_ptrs_def fMax_ge ffmember_filter*
fimage_eqI is_element_ptr_ref)

lemma *new_element_is_l_new_element [instances]:*
"l_new_element type_wf"
using *l_new_element.intro new_element_type_wf_preserved*
by *blast*

lemma *put_M$_{Element}$_tag_type_type_wf_preserved [simp]:*
"h ⊢ put_M element_ptr tag_type_update v →$_h$ h' ⟹ type_wf h = type_wf h'"
apply(*auto simp add: ElementMonad.put_M_defs put$_{Element}$_def put$_{Node}$_def*
DocumentClass.type_wf$_{CharacterData}$ DocumentClass.type_wf$_{Element}$
DocumentClass.type_wf$_{Node}$ DocumentClass.type_wf$_{Object}$
is_node_kind_def
dest!: get_heap_E
elim!: bind_returns_heap_E2
intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
apply(*auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs NodeClass.type_wf_defs*
ElementMonad.get_M_defs ObjectClass.type_wf_defs
CharacterDataClass.type_wf_defs split: option.splits)[1]
apply *(metis (no_types, lifting) Option.bind_cong bind_rzero*
get$_{Element}$_def option.distinct(1))
by *metis*

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'"
apply(*auto simp add: ElementMonad.put_M_defs put$_{Element}$_def put$_{Node}$_def*
DocumentClass.type_wf$_{CharacterData}$ DocumentClass.type_wf$_{Element}$
DocumentClass.type_wf$_{Node}$ DocumentClass.type_wf$_{Object}$
is_node_kind_def
dest!: get_heap_E
elim!: bind_returns_heap_E2
intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
apply(*auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs*
NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
CharacterDataClass.type_wf_defs split: option.splits)[1]
apply *(metis (no_types, lifting) Option.bind_cong bind_rzero get$_{Element}$_def option.distinct(1))*
by *metis*

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'"
apply(*auto simp add: ElementMonad.put_M_defs put$_{Element}$_def put$_{Node}$_def*
DocumentClass.type_wf$_{CharacterData}$ DocumentClass.type_wf$_{Element}$

```
    DocumentClass.type_wf_Node DocumentClass.type_wf_Object
    is_node_kind_def
    dest!: get_heap_E
    elim!: bind_returns_heap_E2
    intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
    NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
    NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
    CharacterDataClass.type_wf_defs split: option.splits)[1]
   apply (metis (no_types, lifting) Option.bind_cong bind_rzero get_Element_def option.distinct(1))
  by metis

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'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
    DocumentClass.type_wf_CharacterData DocumentClass.type_wf_Element
    DocumentClass.type_wf_Node DocumentClass.type_wf_Object
    is_node_kind_def
    dest!: get_heap_E
    elim!: bind_returns_heap_E2
    intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
    NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
    NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
    CharacterDataClass.type_wf_defs split: option.splits)[1]
   apply (metis (no_types, lifting) Option.bind_cong bind_rzero get_Element_def option.distinct(1))
  by metis

lemma new_character_data_type_wf_preserved [simp]:
  "h ⊢ new_character_data →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: ElementMonad.put_M_defs put_Element_def put_Node_def
    DocumentClass.type_wf_CharacterData DocumentClass.type_wf_Element
    DocumentClass.type_wf_Node DocumentClass.type_wf_Object
    is_node_kind_def
    new_character_data_def new_CharacterData_def Let_def put_CharacterData_def put_Node_def
    dest!: get_heap_E
    elim!: bind_returns_heap_E2 bind_returns_heap_E type_wf_put_ptr_not_in_heap_E
    intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
    NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
  by (meson new_CharacterData_def new_CharacterData_ptr_not_in_heap)

lemma new_character_data_is_l_new_character_data [instances]:
  "l_new_character_data type_wf"
  using l_new_character_data.intro new_character_data_type_wf_preserved
  by blast

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'"
  apply(auto simp add: CharacterDataMonad.put_M_defs put_CharacterData_def put_Node_def
    DocumentClass.type_wf_CharacterData DocumentClass.type_wf_Element
    DocumentClass.type_wf_Node DocumentClass.type_wf_Object is_node_kind_def
    dest!: get_heap_E  elim!: bind_returns_heap_E2
    intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I ElementMonad.type_wf_put_I
    NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
  apply(auto simp add: is_node_kind_def type_wf_defs ElementClass.type_wf_defs
    NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
    CharacterDataClass.type_wf_defs split: option.splits)[1]
   apply (metis (no_types, lifting) CharacterDataMonad.a_get_M_def bind_eq_None_conv
    error_returns_result get_CharacterData_def get_heap_returns_result option.exhaust_sel
    option.simps(4))
  by (metis (no_types, lifting) CharacterDataMonad.a_get_M_def error_returns_result
    get_heap_returns_result option.exhaust_sel option.simps(4))
```

```
lemma new_document_type_wf_preserved [simp]: "h ⊢ new_document →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: new_document_def new_Document_def Let_def put_Document_def
      DocumentClass.type_wf_CharacterData  DocumentClass.type_wf_Element
      DocumentClass.type_wf_Node DocumentClass.type_wf_Object
      is_node_ptr_kind_none
      elim!: bind_returns_heap_E type_wf_put_ptr_not_in_heap_E
      intro!: type_wf_put_I ElementMonad.type_wf_put_I CharacterDataMonad.type_wf_put_I
      NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I
      split: if_splits)[1]
   apply(auto simp add: type_wf_defs ElementClass.type_wf_defs CharacterDataClass.type_wf_defs
      NodeClass.type_wf_defs ObjectClass.type_wf_defs is_document_kind_def
      split: option.splits)[1]
  by (meson new_Document_def new_Document_ptr_not_in_heap)

locale l_new_document = l_type_wf +
  assumes new_document_types_preserved: "h ⊢ new_document →_h h' ⟹ type_wf h = type_wf h'"

lemma new_document_is_l_new_document  [instances]: "l_new_document type_wf"
  using l_new_document.intro new_document_type_wf_preserved
  by blast

lemma put_M_Document_doctype_type_wf_preserved [simp]:
  "h ⊢ put_M document_ptr doctype_update v →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: put_M_defs put_Document_def dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I
      ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I ObjectMonad.type_wf_put_I)[1]
         apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
        apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
       apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
      apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
     apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
    apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
   apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
  apply(auto simp add: is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
  apply(auto simp add: get_M_defs)
  by (metis (no_types, lifting)  error_returns_result option.exhaust_sel option.simps(4))

lemma put_M_Document_document_element_type_wf_preserved [simp]:
  "h ⊢ put_M document_ptr document_element_update v →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: put_M_defs put_Document_def
      DocumentClass.type_wf_CharacterData
      DocumentClass.type_wf_Element
      DocumentClass.type_wf_Node
      DocumentClass.type_wf_Object is_node_ptr_kind_none
      cast_Object2Document_none is_document_kind_def
      dest!: get_heap_E
```

```
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I
      ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
  apply(auto simp add: get_M_defs is_document_kind_def type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs
      split: option.splits)[1]
  by (metis)

lemma put_M_Document_disconnected_nodes_type_wf_preserved [simp]:
  "h ⊢ put_M document_ptr disconnected_nodes_update v →_h h' ⟹ type_wf h = type_wf h'"
  apply(auto simp add: put_M_defs put_Document_def
      DocumentClass.type_wf_CharacterData
      DocumentClass.type_wf_Element
      DocumentClass.type_wf_Node
      DocumentClass.type_wf_Object
      is_node_ptr_kind_none
      cast_Object2Document_none is_document_kind_def
      dest!: get_heap_E
      elim!: bind_returns_heap_E2
      intro!: type_wf_put_I CharacterDataMonad.type_wf_put_I
      ElementMonad.type_wf_put_I NodeMonad.type_wf_put_I
      ObjectMonad.type_wf_put_I)[1]
  apply(auto simp add: is_document_kind_def get_M_defs type_wf_defs ElementClass.type_wf_defs
      NodeClass.type_wf_defs ElementMonad.get_M_defs ObjectClass.type_wf_defs
      CharacterDataClass.type_wf_defs split: option.splits)[1]
  by (metis)

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'"
  by(simp add: document_ptr_kinds_def preserved_def object_ptr_kinds_preserved_small[OF assms])

lemma document_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 "document_ptr_kinds h = document_ptr_kinds h'"
  using writes_small_big[OF assms]
  apply(simp add: reflp_def transp_def preserved_def document_ptr_kinds_def)
  by (metis assms object_ptr_kinds_preserved)

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'"
  using type_wf_preserved_small[OF assms(1) assms(2) assms(3) assms(4)]
    allI[OF assms(5), of id, simplified] document_ptr_kinds_small[OF assms(1)]
  apply(auto simp add: type_wf_defs )[1]
   apply(auto simp add: type_wf_defs preserved_def get_M_defs document_ptr_kinds_small[OF assms(1)]
      split: option.splits)[1]
   apply force
  apply(auto simp add: type_wf_defs preserved_def get_M_defs document_ptr_kinds_small[OF assms(1)]
      split: option.splits)[1]
  by force

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'"
  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 -
  have "⋀h h' w. w ∈ SW ⟹ h ⊢ w →_h h' ⟹ DocumentClass.type_wf h = DocumentClass.type_wf h'"
    using assms type_wf_preserved_small by fast
  with assms(1) assms(2) show ?thesis
    apply(rule writes_small_big)
    by(auto simp add: reflp_def transp_def)
qed

lemma type_wf_drop: "type_wf h ⟹ type_wf (Heap (fmdrop ptr (the_heap h)))"
  apply(auto simp add: type_wf_defs)[1]
  using type_wf_drop
   apply blast
  by (metis (mono_tags, lifting) comp_apply document_ptr_kinds_commutes ffmember_filter fmdom_filter
      fmfilter_alt_defs(1) fmlookup_drop get_Document_def get_Object_def heap.sel object_ptr_kinds_def)
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 int he 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)"*
  apply(*induct xs*)
  by(*auto*)

lemma *concat_map_distinct:*

```
  "distinct (concat (map f xs)) ⟹ y ∈ set (concat (map f xs)) ⟹ ∃!x ∈ set xs. y ∈ set (f x)"
  apply(induct xs)
  by(auto)

lemma concat_map_all_distinct: "distinct (concat (map f xs)) ⟹ x ∈ set xs ⟹ distinct (f x)"
  apply(induct xs)
  by(auto)

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)))"
  using assms
  apply(induct xs)
  by(auto)

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)"
  using assms
  apply(induct xs)
  by(auto)

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)"
  using assms
  by(auto simp add: bind_def returns_result_def returns_heap_def is_OK_def execute_def pure_def
              split: sum.splits)
```

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 ())))"
  by(auto simp add: known_ptr_impl get_child_nodes_impl a_get_child_nodes_def a_get_child_nodes_tups_def

                    known_ptr_defs CharacterDataClass.known_ptr_defs ElementClass.known_ptr_defs
                    NodeClass.known_ptr_defs
          split: invoke_splits)

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 ()))))"
  by(auto simp add: known_ptr_impl get_child_nodes_impl a_get_child_nodes_def
                  a_get_child_nodes_tups_def known_ptr_defs CharacterDataClass.known_ptr_defs
                  ElementClass.known_ptr_defs NodeClass.known_ptr_defs
          split: invoke_splits)

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)"
  using assms(1) assms(2) assms(3)
  apply(auto simp add:  known_ptr_impl type_wf_impl get_child_nodes_def a_get_child_nodes_tups_def)[1]
  apply(split invoke_splits, rule conjI)+
     apply((rule impI)+, drule(1) known_ptr_not_document_ptr, drule(1) known_ptr_not_character_data_ptr,

                         drule(1) known_ptr_not_element_ptr)
     apply(auto simp add: NodeClass.known_ptr_defs)[1]
    apply(auto simp add: get_child_nodes_document_ptr_def dest: get_M_Document_ok
              split: list.splits option.splits intro!: bind_is_OK_I2)[1]
   apply(auto simp add: get_child_nodes_character_data_ptr_def)[1]
  apply (auto simp add: get_child_nodes_element_ptr_def  CharacterDataClass.type_wf_defs
                       DocumentClass.type_wf_defs intro!: bind_is_OK_I2 split: option.splits)[1]
  using get_M_Element_ok ‹type_wf h›[unfolded type_wf_impl] by blast

lemma get_child_nodes_ptr_in_heap [simp]:
  assumes "h ⊢ get_child_nodes ptr →r children"
  shows "ptr |∈| object_ptr_kinds h"
  using assms
  by(auto simp add: get_child_nodes_impl a_get_child_nodes_def invoke_ptr_in_heap
         dest: is_OK_returns_result_I)

lemma get_child_nodes_pure [simp]:
  "pure (get_child_nodes ptr) h"
  apply (auto simp add: get_child_nodes_impl a_get_child_nodes_def a_get_child_nodes_tups_def)[1]
  apply(split invoke_splits, rule conjI)+
  by(auto simp add: get_child_nodes_document_ptr_def get_child_nodes_character_data_ptr_def
                  get_child_nodes_element_ptr_def intro!: bind_pure_I split: option.splits)

lemma get_child_nodes_reads: "reads (get_child_nodes_locs ptr) (get_child_nodes ptr) h h'"
  apply(simp add: get_child_nodes_locs_impl get_child_nodes_impl a_get_child_nodes_def
      a_get_child_nodes_tups_def a_get_child_nodes_locs_def)
  apply(split invoke_splits, rule conjI)+
     apply(auto)[1]
    apply(auto simp add: get_child_nodes_document_ptr_def intro: reads_subset[OF reads_singleton]
      reads_subset[OF check_in_heap_reads]
      intro!: reads_bind_pure  reads_subset[OF return_reads] split: option.splits)[1]
   apply(auto simp add: get_child_nodes_character_data_ptr_def intro: reads_subset[OF check_in_heap_reads]

      intro!: reads_bind_pure  reads_subset[OF return_reads] )[1]
  apply(auto simp add: get_child_nodes_element_ptr_def intro: reads_subset[OF reads_singleton]
      reads_subset[OF check_in_heap_reads] intro!: reads_bind_pure  reads_subset[OF return_reads]
      split: option.splits)
  done
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
  .
```

```
interpretation
  i_get_child_nodes?: l_get_child_nodes_Core_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  by(auto simp add: l_get_child_nodes_Core_DOM_def get_child_nodes_def get_child_nodes_locs_def)
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"
  apply(unfold_locales)
  using get_child_nodes_reads get_child_nodes_ok get_child_nodes_ptr_in_heap get_child_nodes_pure
  by blast+
```

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'"
  by (auto simp add: get_child_nodes_locs_def new_element_get_M_Object new_element_get_M_Element
                     new_element_get_M_Document split: prod.splits if_splits option.splits
          elim!: bind_returns_result_E bind_returns_heap_E intro: is_element_ptr_kind_obtains)

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 []"
  apply(auto simp add: get_child_nodes_def a_get_child_nodes_tups_def
      split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)[1]
  apply(split invoke_splits, rule conjI)+
    apply(auto intro: new_element_is_element_ptr)[1]
  by(auto simp add: new_element_ptr_in_heap get_child_nodes_element_ptr_def check_in_heap_def
      new_element_child_nodes intro!: bind_pure_returns_result_I
      intro: new_element_is_element_ptr elim!: new_element_ptr_in_heap)
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
  by(unfold_locales)
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"
  using new_element_is_l_new_element get_child_nodes_is_l_get_child_nodes
  apply(auto simp add: l_new_element_get_child_nodes_def l_new_element_get_child_nodes_axioms_def)[1]
  using get_child_nodes_new_element new_element_no_child_nodes
  by fast+
```

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'"
  by (auto simp add: get_child_nodes_locs_def new_character_data_get_M_Object
                     new_character_data_get_M_Element new_character_data_get_M_Document
           split: prod.splits if_splits option.splits
           elim!: bind_returns_result_E bind_returns_heap_E
           intro: is_character_data_ptr_kind_obtains)

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 []"
  apply(auto simp add: get_child_nodes_def a_get_child_nodes_tups_def
             split: prod.splits elim!: bind_returns_result_E bind_returns_heap_E)[1]
  apply(split invoke_splits, rule conjI)+
     apply(auto intro: new_character_data_is_character_data_ptr)[1]
  by(auto simp add: new_character_data_ptr_in_heap get_child_nodes_character_data_ptr_def
                    check_in_heap_def new_character_data_child_nodes
          intro!: bind_pure_returns_result_I
          intro: new_character_data_is_character_data_ptr elim!: new_character_data_ptr_in_heap)
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_nodes_Core_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  by(unfold_locales)
declare l_new_character_data_get_child_nodes_Core_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"
  using new_character_data_is_l_new_character_data get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_new_character_data_get_child_nodes_def l_new_character_data_get_child_nodes_axioms_def)
  using get_child_nodes_new_character_data new_character_data_no_child_nodes
  by fast

new_document  locale l_new_document_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_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'"
  by (auto simp add: get_child_nodes_locs_def new_document_get_M_Object new_document_get_M_Element
                     new_document_get_M_Document split: prod.splits if_splits option.splits
           elim!: bind_returns_result_E bind_returns_heap_E
           intro: is_document_ptr_kind_obtains)

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 []"
  apply(auto simp add: get_child_nodes_def a_get_child_nodes_tups_def
      split: prod.splits
```

```
      elim!: bind_returns_result_E bind_returns_heap_E)[1]
  apply(split invoke_splits, rule conjI)+
    apply(auto intro: new_document_is_document_ptr)[1]
  by(auto simp add: new_document_ptr_in_heap get_child_nodes_document_ptr_def check_in_heap_def
      new_document_document_element
      intro!: bind_pure_returns_result_I
      intro: new_document_is_document_ptr elim!: new_document_ptr_in_heap split: option.splits)
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_nodes_Core_DOM type_wf known_ptr get_child_nodes get_child_nodes_locs
  by(unfold_locales)
declare l_new_document_get_child_nodes_Core_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"
  using new_document_is_l_new_document get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_new_document_get_child_nodes_def l_new_document_get_child_nodes_axioms_def)
  using get_child_nodes_new_document new_document_no_child_nodes
  by fast
```

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))))"
  by(auto simp add: known_ptr_impl set_child_nodes_impl a_set_child_nodes_def
          a_set_child_nodes_tups_def known_ptr_defs CharacterDataClass.known_ptr_defs
          ElementClass.known_ptr_defs NodeClass.known_ptr_defs split: invoke_splits)

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)))))"
  by(auto simp add: known_ptr_impl set_child_nodes_impl a_set_child_nodes_def
            a_set_child_nodes_tups_def known_ptr_defs CharacterDataClass.known_ptr_defs
            ElementClass.known_ptr_defs NodeClass.known_ptr_defs split: invoke_splits)[1]
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'"
  apply(simp add: set_child_nodes_locs_impl set_child_nodes_impl a_set_child_nodes_def
      a_set_child_nodes_tups_def a_set_child_nodes_locs_def)
  apply(split invoke_splits, rule conjI)+
     apply(auto)[1]
    apply(auto simp add: set_child_nodes_document_ptr_def intro!: writes_bind_pure
      intro: writes_union_right_I split: list.splits)[1]
       apply(auto intro: writes_union_right_I split: option.splits)[1]
      apply(auto intro: writes_union_right_I split: option.splits)[1]
     apply(auto intro: writes_union_right_I split: option.splits)[1]
    apply(auto intro: writes_union_right_I split: option.splits)[1]
```

```
    apply(auto intro: writes_union_right_I split: option.splits)[1]
   apply(auto intro: writes_union_right_I split: option.splits)[1]
  apply(auto simp add: set_child_nodes_character_data_ptr_def intro!: writes_bind_pure)[1]
  apply(auto simp add: set_child_nodes_element_ptr_def intro: writes_union_left_I
      intro!: writes_bind_pure split: list.splits option.splits)[1]
  done

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'"
  using assms(1) object_ptr_kinds_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: set_child_nodes_locs_impl all_args_def a_set_child_nodes_locs_def
        split: if_splits)

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'"
  using assms(1) type_wf_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: set_child_nodes_locs_impl type_wf_impl all_args_def a_set_child_nodes_locs_def
        split: if_splits)
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 .

interpretation
  i_set_child_nodes?: l_set_child_nodes_Core_DOM type_wf known_ptr set_child_nodes set_child_nodes_locs
  apply(unfold_locales)
  by (auto simp add: set_child_nodes_def set_child_nodes_locs_def)
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"
  apply(unfold_locales)
  using set_child_nodes_pointers_preserved set_child_nodes_typess_preserved set_child_nodes_writes
  by blast+

get_child_nodes  locale l_set_child_nodes_get_child_nodes_Core_DOM = l_get_child_nodes_Core_DOM + l_set_child_nodes
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 -
  have "h ⊢ check_in_heap ptr →_r ()"
    using assms set_child_nodes_impl[unfolded a_set_child_nodes_def] invoke_ptr_in_heap
    by (metis (full_types) check_in_heap_ptr_in_heap is_OK_returns_heap_I is_OK_returns_result_E
        old.unit.exhaust)
  then have ptr_in_h: "ptr |∈| object_ptr_kinds h"
```

```
    by (simp add: check_in_heap_ptr_in_heap is_OK_returns_result_I)

  have "type_wf h'"
    apply(unfold type_wf_impl)
    apply(rule subst[where P=id, OF type_wf_preserved[OF set_child_nodes_writes assms(3),
          unfolded all_args_def], simplified])
    by(auto simp add: all_args_def assms(2)[unfolded type_wf_impl]
        set_child_nodes_locs_impl[unfolded a_set_child_nodes_locs_def]
        split: if_splits)
  have "h' ⊢ check_in_heap ptr →_r ()"
    using check_in_heap_reads set_child_nodes_writes assms(3) ⟨h ⊢ check_in_heap ptr →_r ()⟩
    apply(rule reads_writes_separate_forwards)
    by(auto simp add: all_args_def set_child_nodes_locs_impl[unfolded a_set_child_nodes_locs_def])
  then have "ptr |∈| object_ptr_kinds h'"
    using check_in_heap_ptr_in_heap by blast
  with assms ptr_in_h ⟨type_wf h'⟩ show ?thesis
    apply(auto simp add: get_child_nodes_impl set_child_nodes_impl type_wf_impl known_ptr_impl
        a_get_child_nodes_def a_get_child_nodes_tups_def a_set_child_nodes_def
        a_set_child_nodes_tups_def
        del: bind_pure_returns_result_I2
        intro!: bind_pure_returns_result_I2)[1]
    apply(split invoke_splits, rule conjI)
     apply(split invoke_splits, rule conjI)
      apply(split invoke_splits, rule conjI)
       apply(auto simp add: NodeClass.known_ptr_defs
        dest!: known_ptr_not_document_ptr known_ptr_not_character_data_ptr
        known_ptr_not_element_ptr)[1]
      apply(auto simp add: NodeClass.known_ptr_defs
        dest!: known_ptr_not_document_ptr known_ptr_not_character_data_ptr
        known_ptr_not_element_ptr)[1]
      apply(auto simp add: get_child_nodes_document_ptr_def set_child_nodes_document_ptr_def get_M_Document_ok
        split: list.splits option.splits
        intro!: bind_pure_returns_result_I2
        dest: get_M_Document_ok; auto dest: returns_result_eq
        dest!: document_put_get[where getter = document_element])[1]
     apply(auto simp add: get_child_nodes_character_data_ptr_def set_child_nodes_character_data_ptr_def)[1]
    by(auto simp add: get_child_nodes_element_ptr_def set_child_nodes_element_ptr_def dest: element_put_get)
qed

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'"
  using assms
  apply(auto simp add: get_child_nodes_locs_impl set_child_nodes_locs_impl all_args_def
      a_set_child_nodes_locs_def a_get_child_nodes_locs_def
      split: if_splits option.splits )[1]
   apply(rule is_document_ptr_kind_obtains)
    apply(simp)
   apply(rule is_document_ptr_kind_obtains)
    apply(auto)[1]
   apply(auto)[1]
  apply(rule is_element_ptr_kind_obtains)
   apply(auto)[1]
  apply(auto)[1]
  apply(rule is_element_ptr_kind_obtains)
   apply(auto)
  done
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
  by unfold_locales
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"
  using get_child_nodes_is_l_get_child_nodes set_child_nodes_is_l_set_child_nodes
  apply(auto simp add: l_set_child_nodes_get_child_nodes_def l_set_child_nodes_get_child_nodes_axioms_def)[1]
  using set_child_nodes_get_child_nodes apply blast
  using set_child_nodes_get_child_nodes_different_pointers apply metis
  done
```

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"
  by (auto simp add: bind_pure_I get_attribute_impl[unfolded a_get_attribute_def])

lemma get_attribute_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (get_attribute element_ptr k)"
  apply(unfold type_wf_impl)
  unfolding get_attribute_impl[unfolded a_get_attribute_def] using get_M_Element_ok
  by (metis bind_is_OK_pure_I return_ok ElementMonad.get_M_pure)

lemma get_attribute_ptr_in_heap:
  "h ⊢ ok (get_attribute element_ptr k) ⟹ element_ptr |∈| element_ptr_kinds h"
  unfolding get_attribute_impl[unfolded a_get_attribute_def]
```

```
  by (meson DocumentMonad.get_M_Element_ptr_in_heap bind_is_OK_E is_OK_returns_result_I)

lemma get_attribute_reads:
  "reads (get_attribute_locs element_ptr) (get_attribute element_ptr k) h h'"
  by(auto simp add: get_attribute_impl[unfolded a_get_attribute_def]
                    get_attribute_locs_impl[unfolded a_get_attribute_locs_def]
                    reads_insert_writes_set_right
          intro!: reads_bind_pure)
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 .

interpretation
  i_get_attribute?: l_get_attribute_Core_DOM type_wf get_attribute get_attribute_locs
  apply(unfold_locales)
  by (auto simp add: get_attribute_def get_attribute_locs_def)
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"
  apply(unfold_locales)
  using get_attribute_reads get_attribute_ok get_attribute_ptr_in_heap get_attribute_pure
  by blast+
```

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)"
  apply(unfold type_wf_impl)
  unfolding set_attribute_impl[unfolded a_set_attribute_def] using get_M_Element_ok put_M_Element_ok
  by(metis (no_types, lifting) DocumentClass.type_wf_Element ElementMonad.get_M_pure bind_is_OK_E
                               bind_is_OK_pure_I is_OK_returns_result_I)

lemma set_attribute_writes:
  "writes (set_attribute_locs element_ptr) (set_attribute element_ptr k v) h h'"
  by(auto simp add: set_attribute_impl[unfolded a_set_attribute_def]
                    set_attribute_locs_impl[unfolded a_set_attribute_locs_def]
          intro: writes_bind_pure)
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 .
interpretation
  i_set_attribute?: l_set_attribute_Core_DOM type_wf set_attribute set_attribute_locs
  apply(unfold_locales)
  by (auto simp add: set_attribute_def set_attribute_locs_def)
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"
  apply(unfold_locales)
  using set_attribute_ok set_attribute_writes
  by blast+

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"
  by(auto simp add: set_attribute_impl[unfolded a_set_attribute_def]
                    get_attribute_impl[unfolded a_get_attribute_def]
          elim!: bind_returns_heap_E2
          intro!: bind_pure_returns_result_I
          elim: element_put_get)
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
  by(unfold_locales)
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"
  using get_attribute_is_l_get_attribute set_attribute_is_l_set_attribute
  apply(simp add: l_set_attribute_get_attribute_def l_set_attribute_get_attribute_axioms_def)
  using set_attribute_get_attribute
  by blast
```

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'))"
  by(auto simp add: set_attribute_locs_def get_child_nodes_locs_def all_args_def
          intro: element_put_get_preserved[where setter=attrs_update])
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
  by unfold_locales
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"
  using set_attribute_is_l_set_attribute get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_set_attribute_get_child_nodes_def l_set_attribute_get_child_nodes_axioms_def)
  using set_attribute_get_child_nodes
  by blast
```

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)"
  apply(unfold type_wf_impl)
  unfolding get_disconnected_nodes_impl[unfolded a_get_disconnected_nodes_def] using get_M_Document_ok
  by fast

lemma get_disconnected_nodes_ptr_in_heap:
  "h ⊢ ok (get_disconnected_nodes document_ptr) ⟹ document_ptr |∈| document_ptr_kinds h"
  unfolding get_disconnected_nodes_impl[unfolded a_get_disconnected_nodes_def]
  by (simp add: DocumentMonad.get_M_ptr_in_heap)

lemma get_disconnected_nodes_pure [simp]: "pure (get_disconnected_nodes document_ptr) h"
  unfolding get_disconnected_nodes_impl[unfolded a_get_disconnected_nodes_def] by simp

lemma get_disconnected_nodes_reads:
  "reads (get_disconnected_nodes_locs document_ptr) (get_disconnected_nodes document_ptr) h h'"
  by(simp add: get_disconnected_nodes_impl[unfolded a_get_disconnected_nodes_def]
               get_disconnected_nodes_locs_impl[unfolded a_get_disconnected_nodes_locs_def]
               reads_bind_pure reads_insert_writes_set_right)
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 .
interpretation
  i_get_disconnected_nodes?: l_get_disconnected_nodes_Core_DOM type_wf get_disconnected_nodes
                             get_disconnected_nodes_locs
  apply(unfold_locales)
  by (auto simp add: get_disconnected_nodes_def get_disconnected_nodes_locs_def)
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"
  apply(simp add: l_get_disconnected_nodes_def)
  using get_disconnected_nodes_reads get_disconnected_nodes_ok get_disconnected_nodes_ptr_in_heap
        get_disconnected_nodes_pure
  by blast+

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'))"
  by(auto simp add: a_set_child_nodes_locs_def a_get_disconnected_nodes_locs_def all_args_def)
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
  by(unfold_locales)
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"
  using set_child_nodes_is_l_set_child_nodes get_disconnected_nodes_is_l_get_disconnected_nodes
  apply(simp add: l_set_child_nodes_get_disconnected_nodes_def
                  l_set_child_nodes_get_disconnected_nodes_axioms_def)
  using set_child_nodes_get_disconnected_nodes
  by fast
```

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'))"
  by(auto simp add: a_set_attribute_locs_def a_get_disconnected_nodes_locs_def all_args_def)
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
  by(unfold_locales)
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"
  using set_attribute_is_l_set_attribute get_disconnected_nodes_is_l_get_disconnected_nodes
  apply(simp add: l_set_attribute_get_disconnected_nodes_def
                  l_set_attribute_get_disconnected_nodes_axioms_def)
  using set_attribute_get_disconnected_nodes
  by fast
```

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'"
  by(auto simp add: get_disconnected_nodes_locs_def new_element_get_M_Document)
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_nodes_Core_DOM type_wf get_disconnected_nodes
  get_disconnected_nodes_locs
  by unfold_locales
declare l_new_element_get_disconnected_nodes_Core_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"
  by (simp add: get_disconnected_nodes_new_element l_new_element_get_disconnected_nodes_def)
```

new_character_data locale `l_new_character_data_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_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'"
  by(auto simp add: get_disconnected_nodes_locs_def new_character_data_get_M_Document)
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_nodes_Core_DOM type_wf get_disconnected_nodes
                                                       get_disconnected_nodes_locs
  by unfold_locales
declare l_new_character_data_get_disconnected_nodes_Core_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"
  by (simp add: get_disconnected_nodes_new_character_data l_new_character_data_get_disconnected_nodes_def)
```

new_document locale `l_new_document_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_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'"
  by(auto simp add: get_disconnected_nodes_locs_def new_document_get_M_Document)

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 []"
  by(simp add: get_disconnected_nodes_def new_document_disconnected_nodes)

end
```

```
interpretation i_new_document_get_disconnected_nodes?:
  l_new_document_get_disconnected_nodes_Core_DOM type_wf get_disconnected_nodes get_disconnected_nodes_locs
  by unfold_locales
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"
  apply (auto simp add: l_new_document_get_disconnected_nodes_def)[1]
  using get_disconnected_nodes_new_document_different_pointers apply fast
  using new_document_no_disconnected_nodes apply blast
  done
```

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_
lemma set_disconnected_nodes_ok:
  "type_wf h ⟹ document_ptr |∈| document_ptr_kinds h ⟹ h ⊢ ok (set_disconnected_nodes document_ptr
node_ptrs)"
  by (simp add: type_wf_impl put_M_Document_ok set_disconnected_nodes_impl[unfolded a_set_disconnected_nodes_def])

lemma set_disconnected_nodes_ptr_in_heap:
  "h ⊢ ok (set_disconnected_nodes document_ptr disc_nodes) ⟹ document_ptr |∈| document_ptr_kinds h"
  by (simp add: set_disconnected_nodes_impl[unfolded a_set_disconnected_nodes_def]
                DocumentMonad.put_M_ptr_in_heap)
```

```
lemma set_disconnected_nodes_writes:
  "writes (set_disconnected_nodes_locs document_ptr) (set_disconnected_nodes document_ptr disc_nodes) h
h'"
  by(auto simp add: set_disconnected_nodes_impl[unfolded a_set_disconnected_nodes_def]
                    set_disconnected_nodes_locs_impl[unfolded a_set_disconnected_nodes_locs_def]
          intro: writes_bind_pure)

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'"
  using assms(1) object_ptr_kinds_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: all_args_def  set_disconnected_nodes_locs_impl[unfolded
                    a_set_disconnected_nodes_locs_def]
          split: if_splits)

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'"
  using assms(1) type_wf_preserved[OF writes_singleton2 assms(2)]
  apply(unfold type_wf_impl)
  by(auto simp add: all_args_def
                    set_disconnected_nodes_locs_impl[unfolded a_set_disconnected_nodes_locs_def]
          split: if_splits)
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 .
interpretation
  i_set_disconnected_nodes?: l_set_disconnected_nodes_Core_DOM type_wf set_disconnected_nodes
                             set_disconnected_nodes_locs
  apply unfold_locales
  by (auto simp add: set_disconnected_nodes_def set_disconnected_nodes_locs_def)
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"
  apply(simp add: l_set_disconnected_nodes_def)
  using set_disconnected_nodes_ok set_disconnected_nodes_writes set_disconnected_nodes_pointers_preserved

        set_disconnected_nodes_ptr_in_heap set_disconnected_nodes_typess_preserved
  by blast+

get_disconnected_nodes  locale l_set_disconnected_nodes_get_disconnected_nodes_Core_DOM = l_get_disconnected_nodes

                                                        + 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"
  using assms
  by(auto simp add: a_get_disconnected_nodes_def a_set_disconnected_nodes_def)

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'"
  using assms
  by(auto simp add: all_args_def a_set_disconnected_nodes_locs_def a_get_disconnected_nodes_locs_def
            split: if_splits option.splits )
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_nodes_Core_DOM type_wf get_disconnected_nodes
                    get_disconnected_nodes_locs set_disconnected_nodes set_disconnected_nodes_locs
  by unfold_locales
declare l_set_disconnected_nodes_get_disconnected_nodes_Core_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"
  using set_disconnected_nodes_is_l_set_disconnected_nodes get_disconnected_nodes_is_l_get_disconnected_nodes
  apply(simp add: l_set_disconnected_nodes_get_disconnected_nodes_def
                  l_set_disconnected_nodes_get_disconnected_nodes_axioms_def)
  using set_disconnected_nodes_get_disconnected_nodes
        set_disconnected_nodes_get_disconnected_nodes_different_pointers
  by fast+
```

get_child_nodes

```
locale l_set_disconnected_nodes_get_child_nodes_Core_DOM =
  l_set_disconnected_nodes_Core_DOM +
  l_get_child_nodes_Core_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'))"
  by(auto simp add: set_disconnected_nodes_locs_impl[unfolded a_set_disconnected_nodes_locs_def]
                    get_child_nodes_locs_impl[unfolded a_get_child_nodes_locs_def] all_args_def)
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_nodes_Core_DOM
                                             type_wf
```

```
                                        set_disconnected_nodes set_disconnected_nodes_locs
                                        known_ptr get_child_nodes get_child_nodes_locs
  by unfold_locales
declare l_set_disconnected_nodes_get_child_nodes_Core_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"
  using set_disconnected_nodes_is_l_set_disconnected_nodes get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_set_disconnected_nodes_get_child_nodes_def)
  using set_disconnected_nodes_get_child_nodes
  by fast
```

get_tag_name

```
locale l_get_tag_name_Core_DOM_defs
begin
definition a_get_tag_name :: "(_) element_ptr ⇒ (_, tag_type) dom_prog"
  where
    "a_get_tag_name element_ptr = get_M element_ptr tag_type"

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_type)}"
end

locale l_get_tag_name_defs =
  fixes get_tag_name :: "(_) element_ptr ⇒ (_, tag_type) dom_prog"
  fixes get_tag_name_locs :: "(_) element_ptr ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"

locale l_get_tag_name_Core_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_type) 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)"
  apply(unfold type_wf_impl get_tag_name_impl[unfolded a_get_tag_name_def])
  using get_M_Element_ok
  by blast

lemma get_tag_name_pure [simp]: "pure (get_tag_name element_ptr) h"
  unfolding get_tag_name_impl[unfolded a_get_tag_name_def]
  by simp

lemma get_tag_name_ptr_in_heap [simp]:
  assumes "h ⊢ get_tag_name element_ptr →r children"
  shows "element_ptr |∈| element_ptr_kinds h"
  using assms
  by(auto simp add: get_tag_name_impl[unfolded a_get_tag_name_def] get_M_Element_ptr_in_heap
        dest: is_OK_returns_result_I)

lemma get_tag_name_reads: "reads (get_tag_name_locs element_ptr) (get_tag_name element_ptr) h h'"
```

```
  by(simp add: get_tag_name_impl[unfolded a_get_tag_name_def]
               get_tag_name_locs_impl[unfolded a_get_tag_name_locs_def] reads_bind_pure
               reads_insert_writes_set_right)
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 .

interpretation
  i_get_tag_name?: l_get_tag_name_Core_DOM type_wf get_tag_name get_tag_name_locs
  apply(unfold_locales)
  by (auto simp add: get_tag_name_def get_tag_name_locs_def)
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"
  apply(unfold_locales)
  using get_tag_name_reads get_tag_name_ok get_tag_name_ptr_in_heap get_tag_name_pure
  by blast+

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'))"
  by(auto simp add: a_set_disconnected_nodes_locs_def a_get_tag_name_locs_def all_args_def)
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_name_Core_DOM type_wf
                                          set_disconnected_nodes set_disconnected_nodes_locs
                                          get_tag_name get_tag_name_locs
  by unfold_locales
declare l_set_disconnected_nodes_get_tag_name_Core_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"
  using set_disconnected_nodes_is_l_set_disconnected_nodes get_tag_name_is_l_get_tag_name
  apply(simp add: l_set_disconnected_nodes_get_tag_name_def l_set_disconnected_nodes_get_tag_name_axioms_def)
  using set_disconnected_nodes_get_tag_name
  by fast

set_child_nodes locale l_set_child_nodes_get_tag_name_Core_DOM =
  l_set_child_nodes_Core_DOM +
  l_get_tag_name_Core_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'))"
  by(auto simp add: set_child_nodes_locs_def get_tag_name_locs_def all_args_def
          intro: element_put_get_preserved[where getter=tag_type and setter=child_nodes_update])
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_name_Core_DOM type_wf known_ptr
                                   set_child_nodes set_child_nodes_locs get_tag_name get_tag_name_locs
  by unfold_locales
declare l_set_child_nodes_get_tag_name_Core_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"
  using set_child_nodes_is_l_set_child_nodes get_tag_name_is_l_get_tag_name
  apply(simp add: l_set_child_nodes_get_tag_name_def l_set_child_nodes_get_tag_name_axioms_def)
  using set_child_nodes_get_tag_name
  by fast
```

set_tag_type

```
locale l_set_tag_type_Core_DOM_defs
begin

definition a_set_tag_type :: "(_) element_ptr ⇒ tag_type ⇒ (_, unit) dom_prog"
  where
    "a_set_tag_type ptr tag = do {
      m ← get_M ptr attrs;
      put_M ptr tag_type_update tag
    }"
lemmas set_tag_type_defs = a_set_tag_type_def

definition a_set_tag_type_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"
  where
    "a_set_tag_type_locs element_ptr ≡ all_args (put_M element_ptr tag_type_update)"
end

locale l_set_tag_type_defs =
  fixes set_tag_type :: "(_) element_ptr ⇒ tag_type ⇒ (_, unit) dom_prog"
  fixes set_tag_type_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set"

locale l_set_tag_type_Core_DOM =
  l_type_wf type_wf +
  l_set_tag_type_defs set_tag_type set_tag_type_locs +
  l_set_tag_type_Core_DOM_defs
  for type_wf :: "(_) heap ⇒ bool"
  and set_tag_type :: "(_) element_ptr ⇒ char list ⇒ (_, unit) dom_prog"
  and set_tag_type_locs :: "(_) element_ptr ⇒ (_, unit) dom_prog set" +
  assumes type_wf_impl: "type_wf = DocumentClass.type_wf"
  assumes set_tag_type_impl: "set_tag_type = a_set_tag_type"
  assumes set_tag_type_locs_impl: "set_tag_type_locs = a_set_tag_type_locs"
begin

lemma set_tag_type_ok:
  "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_tag_type element_ptr tag)"
  apply(unfold type_wf_impl)
  unfolding set_tag_type_impl[unfolded a_set_tag_type_def] using get_M_Element_ok put_M_Element_ok
  by (metis (no_types, lifting) DocumentClass.type_wf_Element ElementMonad.get_M_pure bind_is_OK_E
```

```
                                bind_is_OK_pure_I is_OK_returns_result_I)

lemma set_tag_type_writes:
  "writes (set_tag_type_locs element_ptr) (set_tag_type element_ptr tag) h h'"
  by(auto simp add: set_tag_type_impl[unfolded a_set_tag_type_def]
                    set_tag_type_locs_impl[unfolded a_set_tag_type_locs_def] intro: writes_bind_pure)

lemma set_tag_type_pointers_preserved:
  assumes "w ∈ set_tag_type_locs element_ptr"
  assumes "h ⊢ w →_h h'"
  shows "object_ptr_kinds h = object_ptr_kinds h'"
  using assms(1) object_ptr_kinds_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: all_args_def set_tag_type_locs_impl[unfolded a_set_tag_type_locs_def]
          split: if_splits)

lemma set_tag_type_typess_preserved:
  assumes "w ∈ set_tag_type_locs element_ptr"
  assumes "h ⊢ w →_h h'"
  shows "type_wf h = type_wf h'"
  apply(unfold type_wf_impl)
  using assms(1) type_wf_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: all_args_def set_tag_type_locs_impl[unfolded a_set_tag_type_locs_def]
          split: if_splits)
end

locale l_set_tag_type = l_type_wf + l_set_tag_type_defs +
  assumes set_tag_type_writes:
    "writes (set_tag_type_locs element_ptr) (set_tag_type element_ptr tag) h h'"
  assumes set_tag_type_ok:
    "type_wf h ⟹ element_ptr |∈| element_ptr_kinds h ⟹ h ⊢ ok (set_tag_type element_ptr tag)"
  assumes set_tag_type_pointers_preserved:
    "w ∈ set_tag_type_locs element_ptr ⟹ h ⊢ w →_h h' ⟹ object_ptr_kinds h = object_ptr_kinds h'"
  assumes set_tag_type_types_preserved:
    "w ∈ set_tag_type_locs element_ptr ⟹ h ⊢ w →_h h' ⟹ type_wf h = type_wf h'"


global_interpretation l_set_tag_type_Core_DOM_defs defines
  set_tag_type = l_set_tag_type_Core_DOM_defs.a_set_tag_type and
  set_tag_type_locs = l_set_tag_type_Core_DOM_defs.a_set_tag_type_locs .
interpretation
  i_set_tag_type?: l_set_tag_type_Core_DOM type_wf set_tag_type set_tag_type_locs
  apply(unfold_locales)
  by (auto simp add: set_tag_type_def set_tag_type_locs_def)
declare l_set_tag_type_Core_DOM_axioms[instances]

lemma set_tag_type_is_l_set_tag_type [instances]:
  "l_set_tag_type type_wf set_tag_type set_tag_type_locs"
  apply(simp add: l_set_tag_type_def)
  using set_tag_type_ok set_tag_type_writes set_tag_type_pointers_preserved
        set_tag_type_typess_preserved
  by blast
```

get_child_nodes

```
locale l_set_tag_type_get_child_nodes_Core_DOM =
  l_set_tag_type_Core_DOM +
  l_get_child_nodes_Core_DOM
begin
lemma set_tag_type_get_child_nodes:
  "∀w ∈ set_tag_type_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"
  by(auto simp add: set_tag_type_locs_impl[unfolded a_set_tag_type_locs_def]
                    get_child_nodes_locs_impl[unfolded a_get_child_nodes_locs_def] all_args_def
          intro: element_put_get_preserved[where setter=tag_type_update and getter=child_nodes])
end
```

```
locale l_set_tag_type_get_child_nodes = l_set_tag_type + l_get_child_nodes +
  assumes set_tag_type_get_child_nodes:
    "∀w ∈ set_tag_type_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_child_nodes_locs ptr'. r h h'))"

interpretation
  i_set_tag_type_get_child_nodes?: l_set_tag_type_get_child_nodes_Core_DOM type_wf
                                   set_tag_type set_tag_type_locs known_ptr
                                   get_child_nodes get_child_nodes_locs
  by unfold_locales
declare l_set_tag_type_get_child_nodes_Core_DOM_axioms[instances]

lemma set_tag_type_get_child_nodes_is_l_set_tag_type_get_child_nodes [instances]:
  "l_set_tag_type_get_child_nodes type_wf set_tag_type set_tag_type_locs known_ptr get_child_nodes
                                  get_child_nodes_locs"
  using set_tag_type_is_l_set_tag_type get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_set_tag_type_get_child_nodes_def l_set_tag_type_get_child_nodes_axioms_def)
  using set_tag_type_get_child_nodes
  by fast
```

get_disconnected_nodes

```
locale l_set_tag_type_get_disconnected_nodes_Core_DOM =
  l_set_tag_type_Core_DOM +
  l_get_disconnected_nodes_Core_DOM
begin
lemma set_tag_type_get_disconnected_nodes:
  "∀w ∈ set_tag_type_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"
  by(auto simp add: set_tag_type_locs_impl[unfolded a_set_tag_type_locs_def]
                    get_disconnected_nodes_locs_impl[unfolded a_get_disconnected_nodes_locs_def]
                    all_args_def)
end

locale l_set_tag_type_get_disconnected_nodes = l_set_tag_type + l_get_disconnected_nodes +
  assumes set_tag_type_get_disconnected_nodes:
   "∀w ∈ set_tag_type_locs ptr. (h ⊢ w →_h h' ⟶ (∀r ∈ get_disconnected_nodes_locs ptr'. r h h'))"

interpretation
  i_set_tag_type_get_disconnected_nodes?: l_set_tag_type_get_disconnected_nodes_Core_DOM type_wf
                                          set_tag_type set_tag_type_locs get_disconnected_nodes
                                          get_disconnected_nodes_locs
  by unfold_locales
declare l_set_tag_type_get_disconnected_nodes_Core_DOM_axioms[instances]

lemma set_tag_type_get_disconnected_nodes_is_l_set_tag_type_get_disconnected_nodes [instances]:
  "l_set_tag_type_get_disconnected_nodes type_wf set_tag_type set_tag_type_locs get_disconnected_nodes
                                         get_disconnected_nodes_locs"
  using set_tag_type_is_l_set_tag_type get_disconnected_nodes_is_l_get_disconnected_nodes
  apply(simp add: l_set_tag_type_get_disconnected_nodes_def
                  l_set_tag_type_get_disconnected_nodes_axioms_def)
  using set_tag_type_get_disconnected_nodes
  by fast
```

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)"
  apply(unfold type_wf_impl)
  unfolding set_val_impl[unfolded a_set_val_def] using get_M_CharacterData_ok put_M_CharacterData_ok
  by (metis (no_types, lifting) DocumentClass.type_wf_CharacterData CharacterDataMonad.get_M_pure
          bind_is_OK_E bind_is_OK_pure_I is_OK_returns_result_I)

lemma set_val_writes: "writes (set_val_locs character_data_ptr) (set_val character_data_ptr tag) h h'"
  by(auto simp add: set_val_impl[unfolded a_set_val_def] set_val_locs_impl[unfolded a_set_val_locs_def]

              intro: writes_bind_pure)

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'"
  using assms(1) object_ptr_kinds_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: all_args_def set_val_locs_impl[unfolded a_set_val_locs_def] split: if_splits)

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'"
  apply(unfold type_wf_impl)
  using assms(1) type_wf_preserved[OF writes_singleton2 assms(2)]
  by(auto simp add: all_args_def set_val_locs_impl[unfolded a_set_val_locs_def] split: if_splits)
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 .
interpretation
  i_set_val?: l_set_val_Core_DOM type_wf set_val set_val_locs
  apply(unfold_locales)
  by (auto simp add: set_val_def set_val_locs_def)
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"
  apply(simp add: l_set_val_def)
  using set_val_ok set_val_writes set_val_pointers_preserved set_val_typess_preserved
  by blast
```

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'))"
  by(auto simp add: set_val_locs_impl[unfolded a_set_val_locs_def]
                    get_child_nodes_locs_impl[unfolded a_get_child_nodes_locs_def] all_args_def)
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
  by unfold_locales
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"
  using set_val_is_l_set_val get_child_nodes_is_l_get_child_nodes
  apply(simp add: l_set_val_get_child_nodes_def l_set_val_get_child_nodes_axioms_def)
  using set_val_get_child_nodes
  by fast
```

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'))"
  by(auto simp add: set_val_locs_impl[unfolded a_set_val_locs_def]
                    get_disconnected_nodes_locs_impl[unfolded a_get_disconnected_nodes_locs_def]
                    all_args_def)
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
  by unfold_locales
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
  using set_val_is_l_set_val get_disconnected_nodes_is_l_get_disconnected_nodes
  apply(simp add: l_set_val_get_disconnected_nodes_def l_set_val_get_disconnected_nodes_axioms_def)
  using set_val_get_disconnected_nodes
  by fast
```

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"
  using get_child_nodes_pure
  by(auto simp add: get_parent_def intro!: bind_pure_I filter_M_pure_I)

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)"
  using assms get_child_nodes_ok get_child_nodes_pure
  by(auto simp add: get_parent_impl[unfolded a_get_parent_def] known_ptrs_known_ptr
```

```
        intro!: bind_is_OK_pure_I filter_M_pure_I filter_M_is_OK_I bind_pure_I)

lemma get_parent_ptr_in_heap [simp]: "h ⊢ ok (get_parent node_ptr) ⟹ node_ptr |∈| node_ptr_kinds h"
  using get_parent_def is_OK_returns_result_I check_in_heap_ptr_in_heap
  by (metis (no_types, lifting) bind_returns_heap_E get_parent_pure node_ptr_kinds_commutes pure_pure)

lemma get_parent_parent_in_heap:
  assumes "h ⊢ get_parent child_node →r Some parent"
  shows "parent |∈| object_ptr_kinds h"
  using assms get_child_nodes_pure
  by(auto simp add: get_parent_def elim!: bind_returns_result_E2
        dest!: filter_M_not_more_elements[where x=parent]
        intro!: filter_M_pure_I bind_pure_I
        split: if_splits)

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"
  using assms get_child_nodes_pure
  by(auto simp add: get_parent_def bind_pure_I
        dest!: filter_M_holds_for_result
        elim!: bind_returns_result_E2
        intro!: filter_M_pure_I
        split: if_splits)

lemma get_parent_reads: "reads get_parent_locs (get_parent node_ptr) h h'"
  using get_child_nodes_reads[unfolded reads_def]
  by(auto simp add: get_parent_def get_parent_locs_def
        intro!: reads_bind_pure reads_subset[OF check_in_heap_reads]
                reads_subset[OF get_child_nodes_reads] reads_subset[OF return_reads]
                reads_subset[OF object_ptr_kinds_M_reads] filter_M_reads filter_M_pure_I bind_pure_I)

lemma get_parent_reads_pointers: "preserved (get_M_Object ptr RObject.nothing) ∈ get_parent_locs"
  by(auto simp add: get_parent_locs_def)
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_M_Object ptr RObject.nothing) ∈ get_parent_locs"

global_interpretation l_get_parent_Core_DOM_defs get_child_nodes get_child_nodes_locs defines
  get_parent = "l_get_parent_Core_DOM_defs.a_get_parent get_child_nodes" and
  get_parent_locs = "l_get_parent_Core_DOM_defs.a_get_parent_locs get_child_nodes_locs" .

interpretation
  i_get_parent?: l_get_parent_Core_DOM known_ptr type_wf get_child_nodes get_child_nodes_locs known_ptrs

                 get_parent get_parent_locs
  using instances
  apply(simp add: l_get_parent_Core_DOM_def l_get_parent_Core_DOM_axioms_def)
```

```
  apply(simp add: get_parent_def get_parent_locs_def)
  done
declare l_get_parent_Core_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"
  using instances
  apply(auto simp add: l_get_parent_def l_get_parent_axioms_def)[1]
  using get_parent_reads get_parent_ok get_parent_ptr_in_heap get_parent_pure
        get_parent_parent_in_heap get_parent_child_dual
  using get_parent_reads_pointers
  by blast+
```

set_disconnected_nodes

```
locale l_set_disconnected_nodes_get_parent_Core_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'))"
  by(auto simp add: get_parent_locs_def set_disconnected_nodes_locs_def all_args_def)
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
  using instances
  by (simp add: l_set_disconnected_nodes_get_parent_Core_DOM_def)
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"
  by(simp add: l_set_disconnected_nodes_get_parent_def)
```

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 -
  have "∀ptr h h' x. h ⊢ get_ancestors ptr →r x ⟶ h ⊢ get_ancestors ptr →h h' ⟶ h = h'"
  proof (induct rule: a_get_ancestors.fixp_induct[folded get_ancestors_impl])
    case 1
    then show ?case
      by(rule admissible_dom_prog)
  next
    case 2
    then show ?case
      by simp
  next
    case (3 f)
    then show ?case
      using get_parent_pure
      apply(auto simp add: pure_returns_heap_eq pure_def
                 split: option.splits
```

```
                elim!: bind_returns_heap_E bind_returns_result_E
                dest!: pure_returns_heap_eq[rotated, OF check_in_heap_pure])[1]
      apply (meson option.simps(3) returns_result_eq)
      by (metis get_parent_pure pure_returns_heap_eq)
  qed
  then show ?thesis
    by (meson pure_eq_iff)
qed


lemma get_root_node_pure [simp]: "pure (get_root_node ptr) h"
  by(auto simp add: get_root_node_def bind_pure_I)


lemma get_ancestors_ptr_in_heap:
  assumes "h ⊢ ok (get_ancestors ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  using assms
  by(auto simp add: get_ancestors_def check_in_heap_ptr_in_heap
          elim!: bind_is_OK_E  dest: is_OK_returns_result_I)

lemma get_ancestors_ptr:
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  shows "ptr ∈ set ancestors"
  using assms
  apply(simp add: get_ancestors_def)
  by(auto  elim!: bind_returns_result_E2 split: option.splits intro!: bind_pure_I)

lemma get_ancestors_not_node:
  assumes "h ⊢ get_ancestors ptr →r ancestors"
  assumes "¬is_node_ptr_kind ptr"
  shows "ancestors = [ptr]"
  using assms
  apply(simp add: get_ancestors_def)
  by(auto elim!: bind_returns_result_E2 split: option.splits)

lemma get_root_node_no_parent:
  "h ⊢ get_parent node_ptr →r None ⟹ h ⊢ get_root_node (cast node_ptr) →r cast node_ptr"
  apply(auto simp add: check_in_heap_def get_root_node_def get_ancestors_def
             intro!: bind_pure_returns_result_I )[1]
  using get_parent_ptr_in_heap by blast

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"
  .
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
  using instances
  apply(simp add: l_get_root_node_Core_DOM_def l_get_root_node_Core_DOM_axioms_def)
  by(simp add: get_root_node_def get_root_node_locs_def get_ancestors_def get_ancestors_locs_def)
declare l_get_root_node_Core_DOM_axioms[instances]

lemma get_ancestors_is_l_get_ancestors [instances]: "l_get_ancestors get_ancestors"
  unfolding l_get_ancestors_def
  using get_ancestors_pure get_ancestors_ptr get_ancestors_ptr_in_heap
  by blast

lemma get_root_node_is_l_get_root_node [instances]: "l_get_root_node get_root_node get_parent"
  apply(simp add: l_get_root_node_def)
  using get_root_node_no_parent
  by fast
```

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'))"
  by(auto simp add: get_parent_locs_def set_disconnected_nodes_locs_def get_ancestors_locs_def
                    all_args_def)
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
  using instances
  by (simp add: l_set_disconnected_nodes_get_ancestors_Core_DOM_def)
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"
  using instances
  apply(simp add: l_set_disconnected_nodes_get_ancestors_def)
  using set_disconnected_nodes_get_ancestors
  by fast
```

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 ())))"
  by(auto simp add: get_owner_document_def a_get_owner_document_tups_def known_ptr_def
                    CharacterDataClass.known_ptr_defs ElementClass.known_ptr_defs
                    NodeClass.known_ptr_defs
          split: invoke_splits option.splits)

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 ()))))"
  by(auto simp add: get_owner_document_def a_get_owner_document_tups_def known_ptr_def
                    CharacterDataClass.known_ptr_defs ElementClass.known_ptr_defs
                    NodeClass.known_ptr_defs
          split: invoke_splits)
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 -
  have "⋀node_ptr. pure (a_get_owner_document_node_ptr node_ptr ()) h"
    by(auto simp add: a_get_owner_document_node_ptr_def
                intro!: bind_pure_I filter_M_pure_I
                split: option.splits)
  moreover have "⋀document_ptr. pure (a_get_owner_document_document_ptr document_ptr ()) h"
    by(auto simp add: a_get_owner_document_document_ptr_def bind_pure_I)
  ultimately show ?thesis
    by(auto simp add: get_owner_document_def a_get_owner_document_tups_def
            intro!: bind_pure_I
            split: invoke_splits)
qed

lemma get_owner_document_ptr_in_heap:
  assumes "h ⊢ ok (get_owner_document ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  using assms
  by(auto simp add: get_owner_document_def invoke_ptr_in_heap dest: is_OK_returns_heap_I)
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"
  .
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
  using instances
    apply(auto simp add: l_get_owner_document_Core_DOM_def l_get_owner_document_Core_DOM_axioms_def)[1]
  by(auto simp add: get_owner_document_tups_def get_owner_document_def get_owner_document_node_ptr_def)[1]
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"
  using get_owner_document_ptr_in_heap
  by(auto simp add: l_get_owner_document_def)
```

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 -
  obtain children where children: "h ⊢ get_child_nodes ptr →r children"
    using assms
    by(auto simp add: remove_child_def)
  moreover have "children ≠ []"
    using assms calculation
    by(auto simp add: remove_child_def elim!: bind_is_OK_E2)
  ultimately show ?thesis
    using assms(1) get_child_nodes_ptr_in_heap by blast
qed

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 -
  obtain prev_disc_nodes h2 children where
    disc_nodes: "h ⊢ get_disconnected_nodes owner_document →r prev_disc_nodes" and
    h2: "h ⊢ set_disconnected_nodes owner_document (child # prev_disc_nodes) →h h2" and
    h': "h2 ⊢ set_child_nodes ptr (remove1 child children) →h h'"
    using assms(1)
```

```
    apply(auto simp add: remove_child_def
              elim!: bind_returns_heap_E
              dest!: returns_result_eq[OF assms(2)] pure_returns_heap_eq[rotated, OF get_owner_document_pure]

                     pure_returns_heap_eq[rotated, OF get_child_nodes_pure]
              split: if_splits)[1]
    by (metis get_disconnected_nodes_pure pure_returns_heap_eq)
  have "h2 ⊢ get_disconnected_nodes owner_document →_r disc_nodes"
    apply(rule reads_writes_separate_backwards[OF get_disconnected_nodes_reads
              set_child_nodes_writes h' assms(3)])
    by (simp add: set_child_nodes_get_disconnected_nodes)
  then show ?thesis
    by (metis (no_types, lifting) h2 set_disconnected_nodes_get_disconnected_nodes
                                  list.set_intros(1) select_result_I2)
qed

lemma remove_child_writes [simp]:
  "writes (remove_child_locs ptr |h ⊢ get_owner_document (cast child)|_r) (remove_child ptr child) h h'"
  apply(auto simp add: remove_child_def intro!: writes_bind_pure[OF get_child_nodes_pure]
                       writes_bind_pure[OF get_owner_document_pure]
                       writes_bind_pure[OF get_disconnected_nodes_pure])[1]
  by(auto simp add: remove_child_locs_def set_disconnected_nodes_writes writes_union_right_I
                    set_child_nodes_writes writes_union_left_I
          intro!: writes_bind)

lemma remove_writes:
  "writes (remove_child_locs (the |h ⊢ get_parent child|_r) |h ⊢ get_owner_document (cast child)|_r) (remove
child) h h'"
  by(auto simp add: remove_def intro!: writes_bind_pure split: option.splits)

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 -
  obtain ptr_children owner_document h2 disc_nodes where
    owner_document: "h ⊢ get_owner_document (cast child) →_r owner_document" and
    ptr_children: "h ⊢ get_child_nodes parent →_r ptr_children" and
    disc_nodes: "h ⊢ get_disconnected_nodes owner_document →_r disc_nodes" and
    h2: "h ⊢ set_disconnected_nodes owner_document (child # disc_nodes) →_h h2" and
    h': "h2 ⊢ set_child_nodes parent (remove1 child ptr_children) →_h h'"
    using assms(1)
    by(auto simp add: remove_child_def
                elim!: bind_returns_heap_E
                dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                       pure_returns_heap_eq[rotated, OF get_disconnected_nodes_pure]
                       pure_returns_heap_eq[rotated, OF get_child_nodes_pure]
                split: if_splits)
  have "parent |∈| object_ptr_kinds h"
    using get_child_nodes_ptr_in_heap ptr_children by blast
  have "object_ptr_kinds h = object_ptr_kinds h2"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF set_disconnected_nodes_writes h2])
    using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  have "type_wf h2"
    using type_wf writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'",
                                    OF set_disconnected_nodes_writes h2]
      using  set_disconnected_nodes_types_preserved
      by(auto simp add: reflp_def transp_def)
```

```
  have "h2 ⊢ get_child_nodes ptr →r children"
   using get_child_nodes_reads set_disconnected_nodes_writes h2 assms(2)
     apply(rule reads_writes_separate_forwards)
     by (simp add: set_disconnected_nodes_get_child_nodes)
  moreover have "h2 ⊢ get_child_nodes parent →r ptr_children"
    using get_child_nodes_reads set_disconnected_nodes_writes h2 ptr_children
     apply(rule reads_writes_separate_forwards)
     by (simp add: set_disconnected_nodes_get_child_nodes)
    moreover have
     "ptr ≠ parent ⟹ h2 ⊢ get_child_nodes ptr →r children = h' ⊢ get_child_nodes ptr →r children"
     using get_child_nodes_reads set_child_nodes_writes h'
     apply(rule reads_writes_preserved)
     by (metis set_child_nodes_get_child_nodes_different_pointers)
  moreover have "h' ⊢ get_child_nodes parent →r remove1 child ptr_children"
    using h' set_child_nodes_get_child_nodes  known_ptrs type_wf known_ptrs_known_ptr
          ⟨parent |∈| object_ptr_kinds h⟩ ⟨object_ptr_kinds h = object_ptr_kinds h2⟩ ⟨type_wf h2⟩
          by fast
  moreover have "set ( remove1 child ptr_children) ⊆ set ptr_children"
    by (simp add: set_remove1_subset)
  ultimately show ?thesis
    by (metis assms(3) order_refl returns_result_eq)
qed


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'"
  using assms
  using set_child_nodes_pointers_preserved
  using set_disconnected_nodes_pointers_preserved
  unfolding remove_child_locs_def
  by auto

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'"
  using assms
  using set_child_nodes_types_preserved
  using set_disconnected_nodes_types_preserved
  unfolding remove_child_locs_def
  by auto
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_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"
  .
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
  using instances
  apply(simp add: l_remove_child_Core_DOM_def l_remove_child_Core_DOM_axioms_def)
  by(simp add: remove_child_def remove_child_locs_def remove_def)
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"
  using instances
  apply(auto simp add: l_remove_child_def l_remove_child_axioms_def)[1]
  using remove_child_pointers_preserved apply(blast)
  using remove_child_pointers_preserved apply(blast)
  using remove_child_types_preserved apply(blast)
  using remove_child_types_preserved apply(blast)
  using remove_child_in_disconnected_nodes apply(blast)
  using remove_child_ptr_in_heap  apply(blast)
  using remove_child_children_subset  apply(blast)
  done
```

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"
  .

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'"
  apply(auto simp add: adopt_node_def adopt_node_locs_def
            intro!: writes_bind_pure[OF get_owner_document_pure] writes_bind_pure[OF get_parent_pure]
                    writes_bind_pure[OF get_disconnected_nodes_pure]
            split: option.splits)[1]
  apply(auto intro!: writes_bind)[1]
  apply (simp add: set_disconnected_nodes_writes writes_union_right_I)
    apply (simp add: set_disconnected_nodes_writes writes_union_left_I writes_union_right_I)
   apply(auto intro!: writes_bind)[1]
  apply (metis (no_types, lifting) remove_child_writes select_result_I2 writes_union_left_I)
    apply (simp add: set_disconnected_nodes_writes writes_union_right_I)
  by(auto intro: writes_subset[OF set_disconnected_nodes_writes] writes_subset[OF remove_child_writes])

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 -
  obtain old_document parent_opt h2 where
    old_document: "h ⊢ get_owner_document (cast node) →_r old_document" and
    parent_opt: "h ⊢ get_parent node →_r parent_opt" and
    h2: "h ⊢ (case parent_opt of Some parent ⇒ do { remove_child parent node } | None ⇒ do { return ()})
→_h h2"
```

```
    and
    h': "h2 ⊢ (if owner_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 owner_document;
        set_disconnected_nodes owner_document (node # disc_nodes)
      } else do { return () }) →_h h'"
    using assms(1)
    by(auto simp add: adopt_node_def
            elim!: bind_returns_heap_E
            dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                   pure_returns_heap_eq[rotated, OF get_parent_pure])

  have "h2 ⊢ get_child_nodes ptr →_r children'"
  proof (cases "owner_document ≠ old_document")
    case True
    then obtain h3 old_disc_nodes disc_nodes where
      old_disc_nodes: "h2 ⊢ get_disconnected_nodes old_document →_r old_disc_nodes" and
      h3: "h2 ⊢ set_disconnected_nodes old_document (remove1 node old_disc_nodes) →_h h3" and
      old_disc_nodes: "h3 ⊢ get_disconnected_nodes owner_document →_r disc_nodes" and
      h': "h3 ⊢ set_disconnected_nodes owner_document (node # disc_nodes) →_h h'"
      using h'
      by(auto elim!: bind_returns_heap_E
                     bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated] )
    have "h3 ⊢ get_child_nodes ptr →_r children'"
      using get_child_nodes_reads set_disconnected_nodes_writes h' assms(3)
      apply(rule reads_writes_separate_backwards)
      by (simp add: set_disconnected_nodes_get_child_nodes)
    show ?thesis
      using get_child_nodes_reads set_disconnected_nodes_writes h3 ⟨h3 ⊢ get_child_nodes ptr →_r children'⟩
      apply(rule reads_writes_separate_backwards)
      by (simp add: set_disconnected_nodes_get_child_nodes)
  next
    case False
    then show ?thesis
      using h' assms(3) by(auto)
  qed

  show ?thesis
  proof (insert h2, induct parent_opt)
    case None
    then show ?case
      using assms
      by(auto dest!: returns_result_eq[OF ⟨h2 ⊢ get_child_nodes ptr →_r children'⟩])
  next
    case (Some option)
    then show ?case
      using assms(2) ⟨h2 ⊢ get_child_nodes ptr →_r children'⟩ remove_child_children_subset known_ptrs type_wf

      by simp
  qed
qed

lemma adopt_node_child_in_heap:
  assumes "h ⊢ ok (adopt_node document_ptr child)"
  shows "child |∈| node_ptr_kinds h"
  using assms
  apply(auto simp add: adopt_node_def elim!: bind_is_OK_E)[1]
  using get_owner_document_pure get_parent_ptr_in_heap pure_returns_heap_eq
  by fast

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'"
  using assms
  using set_disconnected_nodes_pointers_preserved
  using remove_child_pointers_preserved
  unfolding adopt_node_locs_def
  by (auto split: if_splits)

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'"
  using assms
  using remove_child_types_preserved
  using set_disconnected_nodes_types_preserved
  unfolding adopt_node_locs_def
  by (auto split: if_splits)
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
  apply(unfold_locales)
  by(auto simp add: adopt_node_def adopt_node_locs_def)
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"
  using instances
  by (simp add: l_adopt_node_axioms_def adopt_node_child_in_heap adopt_node_children_subset
                adopt_node_pointers_preserved adopt_node_types_preserved adopt_node_writes
                l_adopt_node_def)
```

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"
  by(auto simp add: a_next_sibling_def get_parent_pure intro!: bind_pure_I split: option.splits list.splits)

lemma insert_before_list_in_set: "x ∈ set (insert_before_list v ref xs) ⟷ x = v ∨ x ∈ set xs"
  apply(induct v ref xs rule: insert_before_list.induct)
  by(auto)

lemma insert_before_list_distinct: "x ∉ set xs ⟹ distinct xs ⟹ distinct (insert_before_list x ref
xs)"
  by (induct x ref xs rule: insert_before_list.induct)
     (auto simp add: insert_before_list_in_set)

lemma insert_before_list_subset: "set xs ⊆ set (insert_before_list x ref xs)"
  apply(induct x ref xs rule: insert_before_list.induct)
  by(auto)

lemma insert_before_list_node_in_set: "x ∈ set (insert_before_list x ref xs)"
  apply(induct x ref xs rule: insert_before_list.induct)
  by(auto)

lemma insert_node_writes:
  "writes (set_child_nodes_locs ptr) (a_insert_node ptr new_child reference_child_opt) h h'"
  by(auto simp add: a_insert_node_def set_child_nodes_writes
```

```
          intro!: writes_bind_pure[OF get_child_nodes_pure])

lemma ensure_pre_insertion_validity_pure [simp]:
  "pure (a_ensure_pre_insertion_validity node ptr child) h"
  by(auto simp add: a_ensure_pre_insertion_validity_def
          intro!: bind_pure_I
          split: option.splits)

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 -
  have "h ⊢ a_ensure_pre_insertion_validity node ptr (Some child) →e NotFoundError"
    using assms unfolding insert_before_def a_ensure_pre_insertion_validity_def
    by auto (simp | rule bind_returns_error_I2)+
  then show ?thesis
    unfolding insert_before_def by auto
qed

lemma insert_before_child_in_heap:
  assumes "h ⊢ ok (insert_before ptr node reference_child)"
  shows "node |∈| node_ptr_kinds h"
  using assms
  apply(auto simp add: insert_before_def elim!: bind_is_OK_E)[1]
  by (metis (mono_tags, lifting) ensure_pre_insertion_validity_pure is_OK_returns_heap_I
          l_get_owner_document.get_owner_document_pure local.adopt_node_child_in_heap
          local.l_get_owner_document_axioms next_sibling_pure pure_returns_heap_eq return_pure)

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 -
  fix ptr' children'
  assume a1: "h2 ⊢ get_child_nodes ptr' →r children'"
  then show "distinct children'"
  proof (cases "ptr = ptr'")
    case True
    have "h2 ⊢ get_child_nodes ptr →r (insert_before_list new_child reference_child_opt children)"
      using assms(1) assms(2) apply(auto simp add: a_insert_node_def elim!: bind_returns_heap_E)[1]
      using returns_result_eq set_child_nodes_get_child_nodes known_ptr type_wf
      using pure_returns_heap_eq by fastforce
    then show ?thesis
      using True a1 assms(2) assms(3) assms(4) insert_before_list_distinct returns_result_eq
      by fastforce
  next
    case False
    have "h ⊢ get_child_nodes ptr' →r children'"
      using  get_child_nodes_reads insert_node_writes insert_node a1
      apply(rule reads_writes_separate_backwards)
      by (meson False set_child_nodes_get_child_nodes_different_pointers)
    then show ?thesis
      using assms(4) by blast
  qed
qed
```

```
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'"
  apply(auto simp add: insert_before_def insert_before_locs_def a_insert_node_def
      intro!: writes_bind)[1]
       apply (metis (no_types, hide_lams) ensure_pre_insertion_validity_pure local.adopt_node_writes
                    local.get_owner_document_pure next_sibling_pure pure_returns_heap_eq
                    select_result_I2 sup_commute writes_union_right_I)
      apply (metis (no_types, hide_lams) ensure_pre_insertion_validity_pure next_sibling_pure
                   pure_returns_heap_eq select_result_I2 set_disconnected_nodes_writes
                   writes_union_right_I)
     apply (simp add: set_child_nodes_writes writes_union_left_I writes_union_right_I)
    apply (metis (no_types, hide_lams) adopt_node_writes ensure_pre_insertion_validity_pure
                 get_owner_document_pure pure_returns_heap_eq select_result_I2 writes_union_left_I)
   apply (metis (no_types, hide_lams) ensure_pre_insertion_validity_pure pure_returns_heap_eq
                select_result_I2 set_disconnected_nodes_writes writes_union_right_I)
  by (simp add: set_child_nodes_writes writes_union_left_I writes_union_right_I)
end


locale l_append_child_Core_DOM =
  l_append_child_defs +
  l_append_child_Core_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_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
  defines
    next_sibling = "l_insert_before_Core_DOM_defs.a_next_sibling get_parent get_child_nodes" and
    insert_node = "l_insert_before_Core_DOM_defs.a_insert_node get_child_nodes set_child_nodes" and
    ensure_pre_insertion_validity = "l_insert_before_Core_DOM_defs.a_ensure_pre_insertion_validity
                                                    get_parent get_child_nodes get_ancestors" and
    insert_before = "l_insert_before_Core_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"
  .

global_interpretation l_append_child_Core_DOM_defs insert_before
  defines append_child = "l_append_child_Core_DOM_defs.a_append_child insert_before"
  .

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
  apply(simp add: l_insert_before_Core_DOM_def l_insert_before_Core_DOM_axioms_def instances)
  by (simp add: insert_before_def insert_before_locs_def)
```

```
declare l_insert_before_Core_DOM_axioms[instances]

interpretation i_append_child?: l_append_child_Core_DOM append_child insert_before insert_before_locs
  apply(simp add: l_append_child_Core_DOM_def instances append_child_def)
  done
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_type_defs set_tag_type set_tag_type_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_type ::
      "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
  and set_tag_type_locs ::
      "(_) element_ptr ⇒ ((_) heap, exception, unit) prog set"
begin
definition a_create_element :: "(_) document_ptr ⇒ tag_type ⇒ (_, (_) element_ptr) dom_prog"
  where
    "a_create_element document_ptr tag = do {
      new_element_ptr ← new_element;
      set_tag_type 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_type ⇒ (_, (_) 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_type set_tag_type_locs
  defines
  create_element = "l_create_element_Core_DOM_defs.a_create_element get_disconnected_nodes
                                                                    set_disconnected_nodes set_tag_type"
  .

locale l_create_element_Core_DOM =
  l_create_element_Core_DOM_defs +
  l_create_element_defs +
  assumes create_element_impl: "create_element = a_create_element"
begin
lemmas create_element_def = a_create_element_def[folded create_element_impl]
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_type
                     set_tag_type_locs create_element
  by unfold_locales (simp add: create_element_def)
```

```
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"
  .

locale l_create_character_data_Core_DOM =
  l_create_character_data_Core_DOM_defs +
  l_create_character_data_defs +
  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]
end

locale l_create_character_data = l_create_character_data_defs

interpretation
  i_create_character_data?: l_create_character_data_Core_DOM set_val set_val_locs get_disconnected_nodes

                            get_disconnected_nodes_locs set_disconnected_nodes
                            set_disconnected_nodes_locs create_character_data
  by unfold_locales (simp add: create_character_data_def)
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"
  .

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
  by(simp add: l_create_document_Core_DOM_def)
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" .
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 -
  have "∀ptr h h' x. h ⊢ to_tree_order ptr →_r x ⟶ h ⊢ to_tree_order ptr →_h h' ⟶ h = h'"
  proof (induct rule: a_to_tree_order.fixp_induct[folded to_tree_order_impl])
    case 1
```

```
    then show ?case
      by (rule admissible_dom_prog)
  next
    case 2
    then show ?case
      by simp
  next
    case (3 f)
    then have "⋀x h. pure (f x) h"
      by (metis is_OK_returns_heap_E is_OK_returns_result_E pure_def)
    then have "⋀xs h. pure (map_M f xs) h"
      by(rule map_M_pure_I)
    then show ?case
      by(auto elim!: bind_returns_heap_E2)
  qed
  then show ?thesis
    unfolding pure_def
    by (metis is_OK_returns_heap_E is_OK_returns_result_E)
qed
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
  apply(unfold_locales)
  by (simp add: to_tree_order_def)
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"
  using to_tree_order_pure l_to_tree_order_def by blast
```

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" .

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
  by unfold_locales (simp add: first_in_tree_order_def)
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_name = to_tree_order ptr ≫=
      map_filter_M (λptr. (case cast ptr of
        Some element_ptr ⇒ do {
          this_tag_name ← get_M element_ptr tag_type;
          (if this_tag_name = tag_name 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" .

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"
  using assms
  by(auto simp add: get_element_by_id_def first_in_tree_order_def
              elim!: map_filter_M_pure_E[where y=element_ptr]  bind_returns_result_E2
              dest!: bind_returns_result_E3[rotated, OF assms(2), rotated]
              intro!: map_filter_M_pure map_M_pure_I bind_pure_I
              split: option.splits list.splits if_splits)

lemma get_elements_by_tag_name_pure [simp]: "pure (get_elements_by_tag_name ptr tag_name) h"
  by(auto simp add: get_elements_by_tag_name_def
          intro!: bind_pure_I map_filter_M_pure
          split: option.splits)
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_name) 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
  using instances
  apply(simp add: l_get_element_by_Core_DOM_def l_get_element_by_Core_DOM_axioms_def)
  by(simp add: get_element_by_id_def get_elements_by_class_name_def get_elements_by_tag_name_def)
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"
  apply(unfold_locales)
  using get_element_by_id_result_in_tree_order get_elements_by_tag_name_pure
  by fast+
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. node_ptr |∈| 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)))"


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}"

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 children. (h ⊢ get_child_nodes ptr →_r children)
    ⟶ fset_of_list children |⊆| node_ptr_kinds h)
      ∧ (∀document_ptr disc_node_ptrs. (h ⊢ get_disconnected_nodes document_ptr →_r disc_node_ptrs)
         ⟶ fset_of_list disc_node_ptrs |⊆| 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"
  .

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"
  by(auto simp add: parent_child_rel_def)

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"
  using assms
  apply(auto simp add: parent_child_rel_def is_OK_returns_result_I )[1]
  using get_child_nodes_ptr_in_heap by blast

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"
  using assms parent_child_rel_child_nodes by blast


lemma parent_child_rel_finite: "finite (parent_child_rel h)"
proof -
  have "parent_child_rel h = (⋃ptr ∈ set |h ⊢ object_ptr_kinds_M|_r.
              (⋃child ∈ set |h ⊢ get_child_nodes ptr|_r. {(ptr, cast child)}))"
    by(auto simp add: parent_child_rel_def)
  moreover have "finite (⋃ptr ∈ set |h ⊢ object_ptr_kinds_M|_r.
                  (⋃child ∈ set |h ⊢ get_child_nodes ptr|_r. {(ptr, cast_node_ptr2object_ptr child)}))"
    by simp
  ultimately show ?thesis
    by simp
qed

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)"
  using assms
  apply(auto simp add: a_distinct_lists_def)[1]
proof -
  fix parent_ptr :: "(_) object_ptr"
  assume a1: "parent_ptr |∈| object_ptr_kinds h"
  assume a2: "(⋃x∈fset (object_ptr_kinds h).
             set |h ⊢ get_child_nodes x|_r) ∩ (⋃x∈fset (document_ptr_kinds h).
                 set |h ⊢ get_disconnected_nodes x|_r) = {}"
  assume a3: "h ⊢ get_disconnected_nodes document_ptr →_r disc_nodes"
  assume a4: "node_ptr ∈ set disc_nodes"
  assume a5: "node_ptr ∈ set |h ⊢ get_child_nodes parent_ptr|_r"
  have f6: "parent_ptr ∈ fset (object_ptr_kinds h)"
    using a1 by auto
  have f7: "document_ptr ∈ fset (document_ptr_kinds h)"
    using a3 by (meson fmember.rep_eq get_disconnected_nodes_ptr_in_heap is_OK_returns_result_I)
  have "|h ⊢ get_disconnected_nodes document_ptr|_r = disc_nodes"
    using a3 by simp
  then show False
    using f7 f6 a5 a4 a2 by blast
qed


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 -
  have h1: "distinct (concat (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|_r)
                             |h ⊢ document_ptr_kinds_M|_r))"
    using assms(1)
    by(simp add: a_distinct_lists_def)
  then show ?thesis
    using concat_map_all_distinct[OF h1] assms(2) is_OK_returns_result_I get_disconnected_nodes_ok
    by (metis (no_types, lifting) DocumentMonad.ptr_kinds_ptr_kinds_M
                                  l_get_disconnected_nodes.get_disconnected_nodes_ptr_in_heap
                                  l_get_disconnected_nodes_axioms select_result_I2)
qed

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 (cases "children = []", simp)
  assume "children ≠ []"
  have h1: "distinct (concat ((map (λptr. |h ⊢ get_child_nodes ptr|r) |h ⊢ object_ptr_kinds_M|r)))"
    using assms(1)
    by(simp add: a_distinct_lists_def)
  show ?thesis
    using concat_map_all_distinct[OF h1] assms(2) assms(3)
    by (metis (no_types, lifting) ObjectMonad.ptr_kinds_ptr_kinds_M get_child_nodes_ptr_in_heap
                                  is_OK_returns_result_I select_result_I2)
qed

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"
  using assms
  apply(auto simp add: heap_is_wellformed_def a_all_ptrs_in_heap_def)[1]
  by (meson fset_of_list_elem fset_rev_mp)

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'"
  using assms
proof (auto simp add: heap_is_wellformed_def a_distinct_lists_def)[1]
  fix x :: "(_) node_ptr"
  assume a1: "ptr ≠ ptr'"
  assume a2: "h ⊢ get_child_nodes ptr →r children"
  assume a3: "h ⊢ get_child_nodes ptr' →r children'"
  assume a4: "distinct (concat (map (λptr. |h ⊢ get_child_nodes ptr|r)
                                    (sorted_list_of_set (fset (object_ptr_kinds h)))))"
  have f5: "|h ⊢ get_child_nodes ptr|r = children"
    using a2 by simp
  have "|h ⊢ get_child_nodes ptr'|r = children'"
    using a3 by (meson select_result_I2)
  then have "ptr ∉ set (sorted_list_of_set (fset (object_ptr_kinds h)))
         ∨ ptr' ∉ set (sorted_list_of_set (fset (object_ptr_kinds h)))
         ∨ set children ∩ set children' = {}"
    using f5 a4 a1 by (meson distinct_concat_map_E(1))
  then show False
    using a3 a2 by (metis (no_types) assms(4) finite_fset fmember.rep_eq is_OK_returns_result_I
                                     local.get_child_nodes_ptr_in_heap set_sorted_list_of_set)
qed

lemma parent_child_rel_child:
  "h ⊢ get_child_nodes ptr →r children ⟹ child ∈ set children ⟷ (ptr, cast child) ∈ parent_child_rel
h"
  by (simp add: is_OK_returns_result_I get_child_nodes_ptr_in_heap parent_child_rel_def)

lemma parent_child_rel_acyclic: "heap_is_wellformed h ⟹ acyclic (parent_child_rel h)"
  by (simp add: acyclic_heap_def local.heap_is_wellformed_def)

lemma heap_is_wellformed_disconnected_nodes_distinct:
  "heap_is_wellformed h ⟹ h ⊢ get_disconnected_nodes document_ptr →r disc_nodes ⟹ distinct disc_nodes"
  using distinct_lists_disconnected_nodes local.heap_is_wellformed_def by blast

lemma parent_child_rel_parent_in_heap:
```

```
  "(parent, child_ptr) ∈ parent_child_rel h ⟹ parent |∈| object_ptr_kinds h"
  using local.parent_child_rel_def by blast

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"
  apply(auto simp add: heap_is_wellformed_def parent_child_rel_def a_all_ptrs_in_heap_def)[1]
  using get_child_nodes_ok
  by (meson fin_mono fset_of_list_elem returns_result_select_result)

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"
  by (meson fset_mp fset_of_list_elem local.a_all_ptrs_in_heap_def local.heap_is_wellformed_def)

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'"
  using DocumentMonad.ptr_kinds_ptr_kinds_M concat_append distinct_append distinct_concat_map_E(1)
        is_OK_returns_result_I local.a_distinct_lists_def local.get_disconnected_nodes_ptr_in_heap
        local.heap_is_wellformed_def select_result_I2
proof -
  assume a1: "heap_is_wellformed h"
  assume a2: "h ⊢ get_disconnected_nodes document_ptr →r disc_nodes"
  assume a3: "h ⊢ get_disconnected_nodes document_ptr' →r disc_nodes'"
  assume a4: "set disc_nodes ∩ set disc_nodes' ≠ {}"
  have f5: "|h ⊢ get_disconnected_nodes document_ptr|r = disc_nodes"
    using a2 by (meson select_result_I2)
  have f6: "|h ⊢ get_disconnected_nodes document_ptr'|r = disc_nodes'"
    using a3 by (meson select_result_I2)
  have "⋀nss nssa. ¬ distinct (concat (nss @ nssa)) ∨ distinct (concat nssa::(_) node_ptr list)"
    by (metis (no_types) concat_append distinct_append)
  then have "distinct (concat (map (λd. |h ⊢ get_disconnected_nodes d|r) |h ⊢ document_ptr_kinds_M|r))"
    using a1 local.a_distinct_lists_def local.heap_is_wellformed_def by blast
  then show ?thesis
    using f6 f5 a4 a3 a2 by (meson DocumentMonad.ptr_kinds_ptr_kinds_M distinct_concat_map_E(1)
                                   is_OK_returns_result_I local.get_disconnected_nodes_ptr_in_heap)
qed

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 = {}"
  by (metis (no_types, hide_lams) disjoint_iff_not_equal distinct_lists_no_parent
                                  is_OK_returns_result_I local.get_child_nodes_ptr_in_heap
                                  local.heap_is_wellformed_def select_result_I2)

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)"
  apply(auto simp add: heap_is_wellformed_def a_distinct_lists_def a_owner_document_valid_def)[1]
  by (meson fmember.rep_eq)
lemma heap_is_wellformed_children_distinct:
  "heap_is_wellformed h ⟹ h ⊢ get_child_nodes ptr →r children ⟹ distinct children"
  by (metis (no_types, lifting) ObjectMonad.ptr_kinds_ptr_kinds_M concat_append distinct_append
                                distinct_concat_map_E(2) is_OK_returns_result_I local.a_distinct_lists_def

                                local.get_child_nodes_ptr_in_heap local.heap_is_wellformed_def
                               select_result_I2)
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
  apply(unfold_locales)
  by(auto simp add: heap_is_wellformed_def parent_child_rel_def)
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"
  apply(auto simp add: l_heap_is_wellformed_def)[1]
  using heap_is_wellformed_children_in_heap
                apply blast
  using heap_is_wellformed_disc_nodes_in_heap
               apply blast
  using heap_is_wellformed_one_parent
              apply blast
  using heap_is_wellformed_one_disc_parent
```

```
            apply blast
  using heap_is_wellformed_children_disc_nodes_different
          apply blast
  using heap_is_wellformed_disconnected_nodes_distinct
         apply blast
  using heap_is_wellformed_children_distinct
        apply blast
  using heap_is_wellformed_children_disc_nodes
       apply blast
  using parent_child_rel_child
      apply (blast)
  using parent_child_rel_child
     apply(blast)
  using parent_child_rel_finite
    apply blast
  using parent_child_rel_acyclic
   apply blast
  using parent_child_rel_node_ptr
  apply blast
  using parent_child_rel_parent_in_heap
 apply blast
  using parent_child_rel_child_in_heap
  apply blast
  done
```

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 -
  obtain ptrs where ptrs: "h ⊢ object_ptr_kinds_M →_r ptrs"
    by(simp add: object_ptr_kinds_M_defs)
  then have h1: "ptr ∈ set ptrs"
    using get_child_nodes_ok assms(2) is_OK_returns_result_I
    by (metis (no_types, hide_lams) ObjectMonad.ptr_kinds_ptr_kinds_M
      ⟨⋀thesis. (⋀ptrs. h ⊢ object_ptr_kinds_M →_r ptrs ⟹ thesis) ⟹ thesis⟩
     get_child_nodes_ptr_in_heap returns_result_eq select_result_I2)

  let ?P = "(λptr. get_child_nodes ptr ≫= (λchildren. return (child ∈ set children)))"
  let ?filter = "filter_M ?P ptrs"
```

```
  have "h ⊢ ok ?filter"
    using ptrs type_wf
    using get_child_nodes_ok
    apply(auto intro!: filter_M_is_OK_I bind_is_OK_pure_I get_child_nodes_ok simp add: bind_pure_I)[1]
    using assms(4) local.known_ptrs_known_ptr by blast
  then obtain parent_ptrs where parent_ptrs: "h ⊢ ?filter →r parent_ptrs"
    by auto

  have h5: "∃!x. x ∈ set ptrs ∧ h ⊢ Heap_Error_Monad.bind (get_child_nodes x)
                (λchildren. return (child ∈ set children)) →r True"
    apply(auto intro!: bind_pure_returns_result_I)[1]
    using heap_is_wellformed_one_parent
  proof -
    have "h ⊢ (return (child ∈ set children)::((_) heap, exception, bool) prog) →r True"
      by (simp add: assms(3))
    then show
      "∃z. z ∈ set ptrs ∧ h ⊢ Heap_Error_Monad.bind (get_child_nodes z)
                             (λns. return (child ∈ set ns)) →r True"
      by (metis (no_types) assms(2) bind_pure_returns_result_I2 h1 is_OK_returns_result_I
                           local.get_child_nodes_pure select_result_I2)
  next
    fix x y
    assume 0: "x ∈ set ptrs"
      and 1: "h ⊢ Heap_Error_Monad.bind (get_child_nodes x)
                (λchildren. return (child ∈ set children)) →r True"
      and 2: "y ∈ set ptrs"
      and 3: "h ⊢ Heap_Error_Monad.bind (get_child_nodes y)
                (λchildren. return (child ∈ set children)) →r True"
      and 4: "(⋀h ptr children ptr' children'. heap_is_wellformed h
              ⟹ h ⊢ get_child_nodes ptr →r children ⟹ h ⊢ get_child_nodes ptr' →r children'
              ⟹ set children ∩ set children' ≠ {} ⟹ ptr = ptr')"
    then show "x = y"
      by (metis (no_types, lifting) bind_returns_result_E disjoint_iff_not_equal heap_is_wellformed
                                    return_returns_result)
  qed

  have "child |∈| node_ptr_kinds h"
    using heap_is_wellformed_children_in_heap heap_is_wellformed assms(2) assms(3)
    by fast
  moreover have "parent_ptrs = [ptr]"
    apply(rule filter_M_ex1[OF parent_ptrs h1 h5])
    using ptrs assms(2) assms(3)
    by(auto simp add: object_ptr_kinds_M_defs bind_pure_I intro!: bind_pure_returns_result_I)
  ultimately show ?thesis
    using ptrs parent_ptrs
    by(auto simp add: bind_pure_I get_parent_def
            elim!: bind_returns_result_E2
            intro!: bind_pure_returns_result_I filter_M_pure_I)
qed

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"
  using assms parent_child_rel_child get_parent_child_dual by auto

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 -
```

```
  fix ptr
  have "wf ((parent_child_rel h)⁻¹)"
    by (simp add: assms(1) finite_acyclic_wf_converse parent_child_rel_acyclic parent_child_rel_finite)
  then show "?thesis"
  proof (induct rule: wf_induct_rule)
    case (less parent)
    then show ?case
      using assms child_parent_dual parent_child_rel_parent
      by (meson converse_iff parent_child_rel_child)
  qed
qed

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(insert assms(1), induct rule: heap_wellformed_induct)
  case (step parent)
  then show ?case
  proof(cases "parent |∈| object_ptr_kinds h")
    case True
    then obtain children where children: "h ⊢ get_child_nodes parent →r children"
      using get_child_nodes_ok assms(2) assms(3)
      by (meson is_OK_returns_result_E local.known_ptrs_known_ptr)
    then show ?thesis
    proof (cases "children = []")
      case True
      then show ?thesis
        using children empty_children
        by simp
    next
      case False
      then show ?thesis
        using assms(6) children last_in_set step.hyps by blast
    qed
  next
    case False
    then show ?thesis
      by (simp add: not_in_heap)
  qed
qed

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 -
  fix ptr
  have "wf ((parent_child_rel h))"
    by (simp add: assms(1) local.parent_child_rel_acyclic local.parent_child_rel_finite
              wf_iff_acyclic_if_finite)

  then show "?thesis"
  proof (induct rule: wf_induct_rule)
    case (less child)
    show ?case
      using assms get_parent_child_dual
      by (metis less.hyps parent_child_rel_parent)
  qed
```

```
qed
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
  using instances
  by(simp add: l_get_parent_wf_Core_DOM_def)
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 -
  have object_ptr_kinds_eq3: "object_ptr_kinds h = object_ptr_kinds h'"
    using assms(2) assms(3) object_ptr_kinds_preserved preserved_object_pointers by blast
  then have object_ptr_kinds_eq:
      "⋀ptrs. h ⊢ object_ptr_kinds_M →_r ptrs = h' ⊢ object_ptr_kinds_M →_r ptrs"
    unfolding object_ptr_kinds_M_defs by simp
  then have object_ptr_kinds_eq2: "|h ⊢ object_ptr_kinds_M|_r = |h' ⊢ object_ptr_kinds_M|_r"
    using select_result_eq by force
  then have node_ptr_kinds_eq2: "|h ⊢ node_ptr_kinds_M|_r = |h' ⊢ node_ptr_kinds_M|_r"
    using node_ptr_kinds_M_eq by auto
  then have node_ptr_kinds_eq3: "node_ptr_kinds h = node_ptr_kinds h'"
    by auto
  have document_ptr_kinds_eq2: "|h ⊢ document_ptr_kinds_M|_r = |h' ⊢ document_ptr_kinds_M|_r"
    using object_ptr_kinds_eq2 document_ptr_kinds_M_eq by auto
  then have document_ptr_kinds_eq3: "document_ptr_kinds h = document_ptr_kinds h'"
    by auto
   have children_eq:
    "⋀ptr children. h ⊢ get_child_nodes ptr →_r children = h' ⊢ get_child_nodes ptr →_r children"
    apply(rule reads_writes_preserved[OF get_child_nodes_reads assms(3) assms(2)])
```

```
    using preserved_get_child_nodes by fast
  then have children_eq2: "⋀ptr. h ⊢ get_child_nodes ptr|_r = h' ⊢ get_child_nodes ptr|_r"
    using select_result_eq by force
  have disconnected_nodes_eq:
    "⋀document_ptr disconnected_nodes.
        h ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes
                            = h' ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes"
    apply(rule reads_writes_preserved[OF get_disconnected_nodes_reads assms(3) assms(2)])
    using preserved_get_disconnected_nodes by fast
  then have disconnected_nodes_eq2:
        "⋀document_ptr. h ⊢ get_disconnected_nodes document_ptr|_r
                          = h' ⊢ get_disconnected_nodes document_ptr|_r"
    using select_result_eq by force
  have get_parent_eq: "⋀ptr parent. h ⊢ get_parent ptr →_r parent = h' ⊢ get_parent ptr →_r parent"
    apply(rule reads_writes_preserved[OF get_parent_reads assms(3) assms(2)])
    using preserved_get_child_nodes preserved_object_pointers unfolding get_parent_locs_def by fast
  have "a_acyclic_heap h"
    using heap_is_wellformed by (simp add: heap_is_wellformed_def)
  have "parent_child_rel h' ⊆ parent_child_rel h"
  proof
    fix x
    assume "x ∈ parent_child_rel h'"
    then show "x ∈ parent_child_rel h"
      by(simp add: parent_child_rel_def  children_eq2 object_ptr_kinds_eq3)
  qed
  then have "a_acyclic_heap h'"
    using ‹a_acyclic_heap h› acyclic_heap_def acyclic_subset by blast

  moreover have "a_all_ptrs_in_heap h"
    using heap_is_wellformed by (simp add: heap_is_wellformed_def)
  then have "a_all_ptrs_in_heap h'"
    by(auto simp add: a_all_ptrs_in_heap_def node_ptr_kinds_def node_ptr_kinds_eq2
                      object_ptr_kinds_eq3 children_eq disconnected_nodes_eq)

  moreover have h0: "a_distinct_lists h"
    using heap_is_wellformed by (simp add: heap_is_wellformed_def)
  have h1: "map (λptr. h ⊢ get_child_nodes ptr|_r) (sorted_list_of_set (fset (object_ptr_kinds h)))
        = map (λptr. h' ⊢ get_child_nodes ptr|_r) (sorted_list_of_set (fset (object_ptr_kinds h')))"
    by (simp add: children_eq2 object_ptr_kinds_eq3)
  have h2: "map (λdocument_ptr. h ⊢ get_disconnected_nodes document_ptr|_r)
              (sorted_list_of_set (fset (document_ptr_kinds h)))
         = map (λdocument_ptr. h' ⊢ get_disconnected_nodes document_ptr|_r)
               (sorted_list_of_set (fset (document_ptr_kinds h')))"
    using disconnected_nodes_eq document_ptr_kinds_eq2 select_result_eq by force
  have "a_distinct_lists h'"
    using h0
    by(simp add: a_distinct_lists_def h1 h2)

  moreover have "a_owner_document_valid h"
    using heap_is_wellformed by (simp add: heap_is_wellformed_def)
  then have "a_owner_document_valid h'"
    by(auto simp add: a_owner_document_valid_def children_eq2 disconnected_nodes_eq2
                      object_ptr_kinds_eq3 node_ptr_kinds_eq3 document_ptr_kinds_eq3)
  ultimately show ?thesis
    by (simp add: heap_is_wellformed_def)
qed
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
  using l_get_parent_wf_Core_DOM_axioms l_heap_is_wellformed_Core_DOM_axioms
```

```
  by (simp add: l_get_parent_wf2_Core_DOM_def)

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"
  using known_ptrs_is_l_known_ptrs
  apply(auto simp add: l_get_parent_wf_def l_get_parent_wf_axioms_def)[1]
  using child_parent_dual heap_wellformed_induct heap_wellformed_induct_rev parent_child_rel_parent
  by metis+
```

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'"
  using assms
  by (metis distinct_remove1_removeAll heap_is_wellformed_disconnected_nodes_distinct
            set_disconnected_nodes_get_disconnected_nodes member_remove remove_code(1)
            returns_result_eq)
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
  using instances
  by (simp add: l_set_disconnected_nodes_get_disconnected_nodes_wfCore_DOM_def)
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"
  apply(auto simp add: l_set_disconnected_nodes_get_disconnected_nodes_wf_def
                       l_set_disconnected_nodes_get_disconnected_nodes_wf_axioms_def instances)[1]
  using remove_from_disconnected_nodes_removes apply fast
  done
```

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 (insert assms(1), induct rule: heap_wellformed_induct_rev)
  case (step child)
  then show ?case
    using [[simproc del: Product_Type.unit_eq]] get_parent_reads[unfolded reads_def]
    apply(simp (no_asm) add: get_ancestors_def)
    by(auto simp add: get_ancestors_locs_def reads_subset[OF return_reads] get_parent_reads_pointers
            intro!: reads_bind_pure reads_subset[OF check_in_heap_reads]
                    reads_subset[OF get_parent_reads] reads_subset[OF get_child_nodes_reads]
            split: option.splits)
qed

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 (insert assms(1) assms(2), induct rule: heap_wellformed_induct_rev)
  case (step child)
  then show ?case
    using assms(3) assms(4)
    apply(simp (no_asm) add: get_ancestors_def)
    apply(simp add: assms(1) get_parent_parent_in_heap)
    by(auto intro!: bind_is_OK_pure_I bind_pure_I get_parent_ok split: option.splits)
qed

lemma get_root_node_ptr_in_heap:
  assumes "h ⊢ ok (get_root_node ptr)"
  shows "ptr |∈| object_ptr_kinds h"
  using assms
  unfolding get_root_node_def
  using get_ancestors_ptr_in_heap
  by auto


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)"
  unfolding get_root_node_def
  using assms get_ancestors_ok
  by auto


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
  assume a1: "h ⊢ get_ancestors (cast_{node_ptr2object_ptr} child) →_r cast_{node_ptr2object_ptr} child # parent #
ancestors"
  then have "h ⊢ Heap_Error_Monad.bind (check_in_heap (cast_{node_ptr2object_ptr} child))
        (λ_. Heap_Error_Monad.bind (get_parent child)
            (λx. Heap_Error_Monad.bind (case x of None ⇒ return [] | Some x ⇒ get_ancestors x)
             (λancestors. return (cast_{node_ptr2object_ptr} child # ancestors))))
```

```
        →r cast_node_ptr2object_ptr child # parent # ancestors"
    by(simp add: get_ancestors_def)
  then show "h ⊢ get_ancestors parent →r parent # ancestors"
    using assms(2) apply(auto elim!: bind_returns_result_E2 split: option.splits)[1]
    using returns_result_eq by fastforce
next
  assume "h ⊢ get_ancestors parent →r parent # ancestors"
  then show "h ⊢ get_ancestors (cast_node_ptr2object_ptr child) →r cast_node_ptr2object_ptr child # parent # ancestors"
    using assms(2)
    apply(simp (no_asm) add: get_ancestors_def)
    apply(auto intro!: bind_pure_returns_result_I split: option.splits)[1]
    by (metis (full_types) assms(2) check_in_heap_ptr_in_heap is_OK_returns_result_I
                           local.get_parent_ptr_in_heap node_ptr_kinds_commutes old.unit.exhaust
                           select_result_I)
qed


lemma get_ancestors_never_empty:
  assumes "heap_is_wellformed h"
    and "h ⊢ get_ancestors child →r ancestors"
  shows "ancestors ≠ []"
proof(insert assms(2), induct arbitrary: ancestors rule: heap_wellformed_induct_rev[OF assms(1)])
  case (1 child)
  then show ?case
  proof (induct "cast_object_ptr2node_ptr child")
    case None
    then show ?case
      apply(simp add: get_ancestors_def)
      by(auto elim!: bind_returns_result_E2 split: option.splits)
  next
    case (Some child_node)
    then obtain parent_opt where parent_opt: "h ⊢ get_parent child_node →r parent_opt"
      apply(simp add: get_ancestors_def)
      by(auto elim!: bind_returns_result_E2 split: option.splits)
    with Some show ?case
    proof(induct parent_opt)
      case None
      then show ?case
        apply(simp add: get_ancestors_def)
        by(auto elim!: bind_returns_result_E2 split: option.splits)
    next
      case (Some option)
      then show ?case
        apply(simp add: get_ancestors_def)
        by(auto elim!: bind_returns_result_E2 split: option.splits)
    qed
  qed
qed


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 (insert assms(1) assms(2) assms(3), induct ptr arbitrary: ancestors
              rule: heap_wellformed_induct_rev)
  case (step child)
  have "child |∈| object_ptr_kinds h"
```

```
    using get_ancestors_ptr_in_heap step(2) by auto

  show ?case
  proof (induct "cast_{object_ptr2node_ptr} child")
    case None
    then have "ancestors = [child]"
      using step(2) step(3)
      by(auto simp add: get_ancestors_def elim!: bind_returns_result_E2)
    show ?case
      using step(2) step(3)
      apply(auto simp add: ‹ancestors = [child]›)[1]
      using assms(4) returns_result_eq by fastforce
  next
    case (Some child_node)
    note s1 = Some
    obtain parent_opt where parent_opt: "h ⊢ get_parent child_node →_r parent_opt"
      using ‹child |∈| object_ptr_kinds h› assms(1) Some[symmetric] get_parent_ok[OF type_wf known_ptrs]
      by (metis (no_types, lifting) is_OK_returns_result_E known_ptrs get_parent_ok
                l_get_parent_{Core_DOM}_axioms node_ptr_casts_commute node_ptr_kinds_commutes)
    then show ?case
    proof (induct parent_opt)
      case None
      then have "ancestors = [child]"
        using step(2) step(3) s1
        apply(simp add: get_ancestors_def)
        by(auto elim!: bind_returns_result_E2 split: option.splits dest: returns_result_eq)
      show ?case
        using step(2) step(3)
        apply(auto simp add: ‹ancestors = [child]›)[1]
        using assms(4) returns_result_eq by fastforce
    next
      case (Some parent)
      have "h ⊢ Heap_Error_Monad.bind (check_in_heap child)
          (λ_. Heap_Error_Monad.bind
                 (case cast_{object_ptr2node_ptr} child of None ⇒ return []
                  | Some node_ptr ⇒ Heap_Error_Monad.bind (get_parent node_ptr)
                                                 (λparent_ptr_opt. case parent_ptr_opt of None ⇒ return []
                                                                                        | Some x ⇒ get_ancestors x))
                 (λancestors. return (child # ancestors)))
    →_r  ancestors"
        using step(2)
        by(simp add: get_ancestors_def)
      moreover obtain tl_ancestors where tl_ancestors: "ancestors = child # tl_ancestors"
        using calculation
        by(auto elim!: bind_returns_result_E2 split: option.splits)
      ultimately have "h ⊢ get_ancestors parent →_r tl_ancestors"
        using s1 Some
        by(auto elim!: bind_returns_result_E2 split: option.splits dest: returns_result_eq)
      show ?case
        using step(1)[OF s1[symmetric, simplified] Some ‹h ⊢ get_ancestors parent →_r tl_ancestors›]
              step(3)
        apply(auto simp add: tl_ancestors)[1]
        by (metis assms(4) insert_iff list.simps(15) local.step(2) returns_result_eq tl_ancestors)
    qed
  qed
qed

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 -
  obtain child_ancestors where child_ancestors: "h ⊢ get_ancestors (cast child) →r child_ancestors"
    by (meson assms(1) assms(4) get_ancestors_ok is_OK_returns_result_I known_ptrs
              local.get_parent_ptr_in_heap node_ptr_kinds_commutes returns_result_select_result
              type_wf)
  then have "parent ∈ set child_ancestors"
    apply(simp add: get_ancestors_def)
    by(auto elim!: bind_returns_result_E2 split: option.splits dest!: returns_result_eq[OF assms(4)]
                                                                      get_ancestors_ptr)
  then show ?thesis
    using assms child_ancestors get_ancestors_subset by blast
qed

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 -
  assume 0: "(⋀children ancestor_child.
        h ⊢ get_child_nodes ancestor →r children ⟹
        ancestor_child ∈ set children ⟹ cast_node_ptr2object_ptr ancestor_child ∈ set ancestors
        ⟹ thesis)"
  have "∃child. h ⊢ get_parent child →r Some ancestor ∧ cast child ∈ set ancestors"
  proof (insert assms(1) assms(2) assms(3) assms(4), induct ptr arbitrary: ancestors
                rule: heap_wellformed_induct_rev)
    case (step child)
    have "child |∈| object_ptr_kinds h"
      using get_ancestors_ptr_in_heap step(4) by auto
    show ?case
    proof (induct "cast_object_ptr2node_ptr child")
      case None
      then have "ancestors = [child]"
        using step(3) step(4)
        by(auto simp add: get_ancestors_def elim!: bind_returns_result_E2)
      show ?case
        using step(2) step(3) step(4)
        by(auto simp add: ‹ancestors = [child]›)
    next
      case (Some child_node)
      note s1 = Some
      obtain parent_opt where parent_opt: "h ⊢ get_parent child_node →r parent_opt"
        using ‹child |∈| object_ptr_kinds h› assms(1) Some[symmetric]
        using get_parent_ok known_ptrs type_wf
        by (metis (no_types, lifting) is_OK_returns_result_E node_ptr_casts_commute
                  node_ptr_kinds_commutes)
      then show ?case
      proof (induct parent_opt)
        case None
        then have "ancestors = [child]"
          using step(2) step(3) step(4) s1
          apply(simp add: get_ancestors_def)
          by(auto elim!: bind_returns_result_E2 split: option.splits dest: returns_result_eq)
        show ?case
          using step(2) step(3) step(4)
          by(auto simp add: ‹ancestors = [child]›)
      next
```

```
      case (Some parent)
      have "h ⊢ Heap_Error_Monad.bind (check_in_heap child)
          (λ_. Heap_Error_Monad.bind
                 (case cast_{object_ptr2node_ptr} child of None ⇒ return []
                  | Some node_ptr ⇒ Heap_Error_Monad.bind (get_parent node_ptr)
                     (λparent_ptr_opt. case parent_ptr_opt of None ⇒ return []
                                                            | Some x ⇒ get_ancestors x))
                 (λancestors. return (child # ancestors)))
        →_r  ancestors"
        using step(4)
        by(simp add: get_ancestors_def)
      moreover obtain tl_ancestors where tl_ancestors: "ancestors = child # tl_ancestors"
        using calculation
        by(auto elim!: bind_returns_result_E2 split: option.splits)
      ultimately have "h ⊢ get_ancestors parent →_r tl_ancestors"
        using s1 Some
        by(auto elim!: bind_returns_result_E2 split: option.splits dest: returns_result_eq)

      have "ancestor ∈ set tl_ancestors"
        using tl_ancestors step(2) step(3) by auto
      show ?case
      proof (cases "ancestor ≠ parent")
        case True
        show ?thesis
          using step(1)[OF s1[symmetric, simplified] Some True
                      ‹ancestor ∈ set tl_ancestors› ‹h ⊢ get_ancestors parent →_r tl_ancestors›]
          using tl_ancestors by auto
      next
        case False
        have "child ∈ set ancestors"
          using step(4) get_ancestors_ptr by simp
        then show ?thesis
          using Some False s1[symmetric] by(auto)
      qed
    qed
  qed
  then obtain child where child: "h ⊢ get_parent child →_r Some ancestor"
       and in_ancestors: "cast child ∈ set ancestors"
    by auto
  then obtain children where
    children: "h ⊢ get_child_nodes ancestor →_r children" and
    child_in_children: "child ∈ set children"
    using get_parent_child_dual by blast
  show thesis
    using 0[OF children child_in_children] child assms(3) in_ancestors by blast
qed

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 (safe)
  assume 3: "(ptr, child) ∈ (parent_child_rel h)*"
  show "ptr ∈ set ancestors"
  proof (insert 3, induct ptr rule: heap_wellformed_induct[OF assms(1)])
    case (1 ptr)
    then show ?case
    proof (cases "ptr = child")
      case True
      then show ?thesis
```

```
      by (metis (no_types, lifting) assms(2) bind_returns_result_E get_ancestors_def
           in_set_member member_rec(1) return_returns_result)
  next
    case False
    obtain ptr_child where
      ptr_child: "(ptr, ptr_child) ∈ (parent_child_rel h) ∧ (ptr_child, child) ∈ (parent_child_rel h)^*"
      using converse_rtranclE[OF 1(2)] ‹ptr ≠ child›
      by metis
    then obtain ptr_child_node
        where ptr_child_ptr_child_node: "ptr_child = cast_{node_ptr2object_ptr} ptr_child_node"
      using ptr_child node_ptr_casts_commute3 parent_child_rel_node_ptr
      by (metis )
    then obtain children where
      children: "h ⊢ get_child_nodes ptr →_r children" and
      ptr_child_node: "ptr_child_node ∈ set children"
    proof -
      assume a1: "⋀children. ⟦h ⊢ get_child_nodes ptr →_r children; ptr_child_node ∈ set children⟧

                  ⟹ thesis"

      have "ptr |∈| object_ptr_kinds h"
        using local.parent_child_rel_parent_in_heap ptr_child by blast
      moreover have "ptr_child_node ∈ set |h ⊢ get_child_nodes ptr|_r"
        by (metis calculation known_ptrs local.get_child_nodes_ok local.known_ptrs_known_ptr
                  local.parent_child_rel_child ptr_child ptr_child_ptr_child_node
                  returns_result_select_result type_wf)
      ultimately show ?thesis
        using a1 get_child_nodes_ok type_wf known_ptrs
        by (meson local.known_ptrs_known_ptr returns_result_select_result)
    qed
    moreover have "(cast_{node_ptr2object_ptr} ptr_child_node, child) ∈ (parent_child_rel h)^*"
      using ptr_child ptr_child_ptr_child_node by auto
    ultimately have "cast_{node_ptr2object_ptr} ptr_child_node ∈ set ancestors"
      using 1 by auto
    moreover have "h ⊢ get_parent ptr_child_node →_r Some ptr"
      using assms(1) children ptr_child_node child_parent_dual
      using known_ptrs type_wf by blast
    ultimately show ?thesis
      using get_ancestors_also_parent assms type_wf by blast
  qed
qed
next
assume 3: "ptr ∈ set ancestors"
show "(ptr, child) ∈ (parent_child_rel h)^*"
proof (insert 3, induct ptr rule: heap_wellformed_induct[OF assms(1)])
  case (1 ptr)
  then show ?case
  proof (cases "ptr = child")
    case True
    then show ?thesis
      by simp
  next
    case False
    then obtain children ptr_child_node where
      children: "h ⊢ get_child_nodes ptr →_r children" and
      ptr_child_node: "ptr_child_node ∈ set children" and
      ptr_child_node_in_ancestors: "cast ptr_child_node ∈ set ancestors"
      using 1(2) assms(2) get_ancestors_obtains_children assms(1)
      using known_ptrs type_wf by blast
    then have "(cast_{node_ptr2object_ptr} ptr_child_node, child) ∈ (parent_child_rel h)^*"
      using 1(1) by blast

    moreover have "(ptr, cast ptr_child_node) ∈ parent_child_rel h"
```

```
      using children ptr_child_node assms(1) parent_child_rel_child_nodes2
      using child_parent_dual known_ptrs parent_child_rel_parent type_wf
      by blast

    ultimately show ?thesis
      by auto
  qed
 qed
qed

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)*"
  using assms get_ancestors_parent_child_rel
  apply(auto simp add: get_root_node_def elim!: bind_returns_result_E2)[1]
  using get_ancestors_never_empty last_in_set by blast


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 -
  have object_ptr_kinds_eq3: "object_ptr_kinds h = object_ptr_kinds h'"
    using pointers_preserved object_ptr_kinds_preserved_small by blast
  then have object_ptr_kinds_M_eq:
        "⋀ptrs. h ⊢ object_ptr_kinds_M →r ptrs = h' ⊢ object_ptr_kinds_M →r ptrs"
    by(simp add: object_ptr_kinds_M_defs)
  then have object_ptr_kinds_eq: "|h ⊢ object_ptr_kinds_M|r = |h' ⊢ object_ptr_kinds_M|r"
    by(simp)
  have "h' ⊢ ok (get_ancestors ptr)"
    using get_ancestors_ok get_ancestors_ptr_in_heap object_ptr_kinds_eq3 assms(1) known_ptrs
      known_ptrs' assms(2) assms(7) type_wf'
    by blast
  then obtain ancestors' where ancestors': "h' ⊢ get_ancestors ptr →r ancestors'"
    by auto

  obtain root where root: "h ⊢ get_root_node ptr →r root"
  proof -
    assume 0: "(⋀root. h ⊢ get_root_node ptr →r root ⟹ thesis)"
    show thesis
      apply(rule 0)
      using assms(7)
      by(auto simp add: get_root_node_def elim!: bind_returns_result_E2 split: option.splits)
  qed

  have children_eq:
   "⋀p children. p ≠ ptr ⟹ h ⊢ get_child_nodes p →r children = h' ⊢ get_child_nodes p →r children"
    using get_child_nodes_reads assms(3)
    apply(simp add: reads_def reflp_def transp_def preserved_def)
    by blast

  have "acyclic (parent_child_rel h)"
```

```
  using assms(1) local.parent_child_rel_acyclic by auto
have "acyclic (parent_child_rel h')"
  using assms(2) local.parent_child_rel_acyclic by blast
have 2: "⋀c parent_opt. cast_node_ptr2object_ptr c ∈ set ancestors ∩ set ancestors'
      ⟹ h ⊢ get_parent c →_r parent_opt = h' ⊢ get_parent c →_r parent_opt"
proof -
  fix c parent_opt
  assume 1: " cast_node_ptr2object_ptr c ∈ set ancestors ∩ set ancestors'"

  obtain ptrs where ptrs: "h ⊢ object_ptr_kinds_M →_r ptrs"
    by simp

  let ?P = "(λptr. Heap_Error_Monad.bind (get_child_nodes ptr) (λchildren. return (c ∈ set children)))"
  have children_eq_True: "⋀p. p ∈ set ptrs ⟹ h ⊢ ?P p →_r True ⟷ h' ⊢ ?P p →_r True"
  proof -
    fix p
    assume "p ∈ set ptrs"
    then show "h ⊢ ?P p →_r True ⟷ h' ⊢ ?P p →_r True"
    proof (cases "p = ptr")
      case True
      have "(cast_node_ptr2object_ptr c, ptr) ∈ (parent_child_rel h)*"
        using get_ancestors_parent_child_rel 1 assms by blast
      then have "(ptr, cast_node_ptr2object_ptr c) ∉ (parent_child_rel h)"
      proof (cases "cast c = ptr")
        case True
        then show ?thesis
          using ‹acyclic (parent_child_rel h)› by(auto simp add: acyclic_def)
      next
        case False
        then have "(ptr, cast_node_ptr2object_ptr c) ∉ (parent_child_rel h)*"
          using ‹acyclic (parent_child_rel h)› False rtrancl_eq_or_trancl rtrancl_trancl_trancl
                ‹(cast_node_ptr2object_ptr c, ptr) ∈ (parent_child_rel h)*›
          by (metis acyclic_def)
        then show ?thesis
          using r_into_rtrancl by auto
      qed
      obtain children where children: "h ⊢ get_child_nodes ptr →_r children"
        using type_wf
        by (metis ‹h' ⊢ ok get_ancestors ptr› assms(1) get_ancestors_ptr_in_heap get_child_nodes_ok
                  heap_is_wellformed_def is_OK_returns_result_E known_ptrs local.known_ptrs_known_ptr

                  object_ptr_kinds_eq3)
      then have "c ∉ set children"
        using ‹(ptr, cast_node_ptr2object_ptr c) ∉ (parent_child_rel h)› assms(1)
        using parent_child_rel_child_nodes2
        using child_parent_dual known_ptrs parent_child_rel_parent
        type_wf by blast
      with children have "h ⊢ ?P p →_r False"
        by(auto simp add: True)

      moreover have "(cast_node_ptr2object_ptr c, ptr) ∈ (parent_child_rel h')*"
        using get_ancestors_parent_child_rel assms(2) ancestors' 1 known_ptrs' type_wf
        type_wf' by blast
      then have "(ptr, cast_node_ptr2object_ptr c) ∉ (parent_child_rel h')"
      proof (cases "cast c = ptr")
        case True
        then show ?thesis
          using ‹acyclic (parent_child_rel h')› by(auto simp add: acyclic_def)
      next
        case False
        then have "(ptr, cast_node_ptr2object_ptr c) ∉ (parent_child_rel h')*"
          using ‹acyclic (parent_child_rel h')› False rtrancl_eq_or_trancl rtrancl_trancl_trancl
                ‹(cast_node_ptr2object_ptr c, ptr) ∈ (parent_child_rel h')*›
```

```
      by (metis acyclic_def)
    then show ?thesis
      using r_into_rtrancl by auto
  qed
  then have "(ptr, cast_{node_ptr2object_ptr} c) ∉ (parent_child_rel h')"
    using r_into_rtrancl by auto
  obtain children' where children': "h' ⊢ get_child_nodes ptr →_r children'"
    using  type_wf  type_wf'
    by (meson ‹h' ⊢ ok (get_ancestors ptr)› assms(2) get_ancestors_ptr_in_heap
                  get_child_nodes_ok is_OK_returns_result_E known_ptrs'
                  local.known_ptrs_known_ptr)
  then have "c ∉ set children'"
    using ‹(ptr, cast_{node_ptr2object_ptr} c) ∉ (parent_child_rel h')› assms(2)  type_wf  type_wf'
    using parent_child_rel_child_nodes2 child_parent_dual known_ptrs' parent_child_rel_parent
    by auto
  with children' have "h' ⊢ ?P p →_r False"
    by(auto simp add: True)

  ultimately show ?thesis
    by (metis returns_result_eq)
next
  case False
  then show ?thesis
    using children_eq ptrs
    by (metis (no_types, lifting) bind_pure_returns_result_I bind_returns_result_E
        get_child_nodes_pure return_returns_result)
qed
qed
have "⋀pa. pa ∈ set ptrs ⟹ h ⊢ ok (get_child_nodes pa
              ≫= (λchildren. return (c ∈ set children))) = h' ⊢ ok ( get_child_nodes pa
              ≫= (λchildren. return (c ∈ set children)))"
  using assms(1) assms(2) object_ptr_kinds_eq ptrs type_wf  type_wf'
  by (metis (no_types, lifting) ObjectMonad.ptr_kinds_ptr_kinds_M bind_is_OK_pure_I
                          get_child_nodes_ok get_child_nodes_pure known_ptrs'
                          local.known_ptrs_known_ptr return_ok select_result_I2)
have children_eq_False:
    "⋀pa. pa ∈ set ptrs ⟹ h ⊢ get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children)) →_r False = h' ⊢ get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children)) →_r False"
proof
  fix pa
  assume "pa ∈ set ptrs"
     and "h ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False"
  have "h ⊢ ok (get_child_nodes pa ≫= (λchildren. return (c ∈ set children)))
       ⟹ h' ⊢ ok ( get_child_nodes pa ≫= (λchildren. return (c ∈ set children)))"
    using ‹pa ∈ set ptrs› ‹⋀pa. pa ∈ set ptrs ⟹ h ⊢ ok (get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children))) = h' ⊢ ok ( get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children)))›
    by auto
  moreover have "h ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False
    ⟹ h' ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False"
    by (metis (mono_tags, lifting) ‹⋀pa. pa ∈ set ptrs
        ⟹ h ⊢ get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children)) →_r True = h' ⊢ get_child_nodes pa
          ≫= (λchildren. return (c ∈ set children)) →_r True› ‹pa ∈ set ptrs›
        calculation is_OK_returns_result_I returns_result_eq returns_result_select_result)
  ultimately show "h' ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False"
    using ‹h ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False›
    by auto
next
  fix pa
  assume "pa ∈ set ptrs"
    and "h' ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →_r False"
```

```
    have "h' ⊢ ok (get_child_nodes pa ≫= (λchildren. return (c ∈ set children)))
          ⟹ h ⊢ ok ( get_child_nodes pa ≫= (λchildren. return (c ∈ set children)))"
      using ⟨pa ∈ set ptrs⟩ ⟨⋀pa. pa ∈ set ptrs
            ⟹ h ⊢ ok (get_child_nodes pa
                ≫= (λchildren. return (c ∈ set children))) = h' ⊢ ok ( get_child_nodes pa
                ≫= (λchildren. return (c ∈ set children)))⟩
      by auto
    moreover have "h' ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →r False
                ⟹ h ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →r False"
      by (metis (mono_tags, lifting)
           ⟨⋀pa. pa ∈ set ptrs ⟹ h ⊢ get_child_nodes pa
                     ≫= (λchildren. return (c ∈ set children)) →r True = h' ⊢ get_child_nodes pa
                     ≫= (λchildren. return (c ∈ set children)) →r True⟩ ⟨pa ∈ set ptrs⟩
            calculation is_OK_returns_result_I returns_result_eq returns_result_select_result)
    ultimately show "h ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →r False"
      using ⟨h' ⊢ get_child_nodes pa ≫= (λchildren. return (c ∈ set children)) →r False⟩ by blast
  qed

  have filter_eq: "⋀xs. h ⊢ filter_M ?P ptrs →r xs = h' ⊢ filter_M ?P ptrs →r xs"
  proof (rule filter_M_eq)
    show
    "⋀xs x. pure (Heap_Error_Monad.bind (get_child_nodes x) (λchildren. return (c ∈ set children))) h"
      by(auto intro!: bind_pure_I)
  next
    show
    "⋀xs x. pure (Heap_Error_Monad.bind (get_child_nodes x) (λchildren. return (c ∈ set children))) h'"
      by(auto intro!: bind_pure_I)
  next
    fix xs b x
    assume 0: "x ∈ set ptrs"
    then show "h ⊢ Heap_Error_Monad.bind (get_child_nodes x) (λchildren. return (c ∈ set children))
→r b
          = h' ⊢ Heap_Error_Monad.bind (get_child_nodes x) (λchildren. return (c ∈ set children)) →r
b"
      apply(induct b)
      using children_eq_True apply blast
      using children_eq_False apply blast
      done
  qed

  show "h ⊢ get_parent c →r parent_opt = h' ⊢ get_parent c →r parent_opt"
    apply(simp add: get_parent_def)
    apply(rule bind_cong_2)
       apply(simp)
      apply(simp)
     apply(simp add: check_in_heap_def node_ptr_kinds_def object_ptr_kinds_eq3)
    apply(rule bind_cong_2)
       apply(auto simp add: object_ptr_kinds_M_eq object_ptr_kinds_eq3)[1]
      apply(auto simp add: object_ptr_kinds_M_eq object_ptr_kinds_eq3)[1]
     apply(auto simp add: object_ptr_kinds_M_eq object_ptr_kinds_eq3)[1]
    apply(rule bind_cong_2)
       apply(auto intro!: filter_M_pure_I bind_pure_I)[1]
      apply(auto intro!: filter_M_pure_I bind_pure_I)[1]
     apply(auto simp add: filter_eq )
    using filter_eq ptrs apply auto[1]
    using filter_eq ptrs by auto
qed

have "ancestors = ancestors'"
proof(insert assms(1) assms(7) ancestors' 2, induct ptr arbitrary: ancestors ancestors'
            rule: heap_wellformed_induct_rev)
  case (step child)
  show ?case
```

```
    using step(2) step(3) step(4)
    apply(simp add: get_ancestors_def)
    apply(auto intro!: elim!: bind_returns_result_E2 split: option.splits)[1]
    using returns_result_eq apply fastforce
    apply (meson option.simps(3) returns_result_eq)
    by (metis IntD1 IntD2 option.inject returns_result_eq step.hyps)
  qed
  then show ?thesis
    using assms(5) ancestors'
    by simp
qed

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 -
  have object_ptr_kinds_M_eq:
    "⋀ptrs. h ⊢ object_ptr_kinds_M →r ptrs = h' ⊢ object_ptr_kinds_M →r ptrs"
    using object_ptr_kinds_eq3
    by(simp add: object_ptr_kinds_M_defs)
  then have object_ptr_kinds_eq: "|h ⊢ object_ptr_kinds_M|r = |h' ⊢ object_ptr_kinds_M|r"
    by(simp)

  show ?thesis
  proof (insert assms(1) assms(3) assms(4) assms(6), induct ptr arbitrary: ancestors ancestors'
        rule: heap_wellformed_induct_rev)
    case (step child)
    have 1: "⋀p parent. h' ⊢ get_parent p →r Some parent ⟹ h ⊢ get_parent p →r Some parent"
    proof -
      fix p parent
      assume "h' ⊢ get_parent p →r Some parent"
      then obtain children' where
        children': "h' ⊢ get_child_nodes parent →r children'" and
        p_in_children': "p ∈ set children'"
        using get_parent_child_dual by blast
      obtain children where children: "h ⊢ get_child_nodes parent →r children"
        using get_child_nodes_ok assms(1) get_child_nodes_ptr_in_heap object_ptr_kinds_eq children'
              known_ptrs
        using  type_wf type_wf'
        by (metis ⟨h' ⊢ get_parent p →r Some parent⟩ get_parent_parent_in_heap is_OK_returns_result_E
                  local.known_ptrs_known_ptr object_ptr_kinds_eq3)
      have "p ∈ set children"
        using assms(5) children children' p_in_children'
        by blast
      then show "h ⊢ get_parent p →r Some parent"
        using child_parent_dual assms(1) children known_ptrs  type_wf  by blast
    qed
    have "node ≠ child"
      using assms(1) get_ancestors_parent_child_rel step.prems(1) step.prems(3) known_ptrs
      using type_wf type_wf'
      by blast
    then show ?case
      using step(2) step(3)
```

```
      apply(simp add: get_ancestors_def)
      using step(4)
      apply(auto elim!: bind_returns_result_E2 split: option.splits)[1]
      using 1
       apply (meson option.distinct(1) returns_result_eq)
      by (metis "1" option.inject returns_result_eq step.hyps)
  qed
qed

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 (insert assms(4) assms(5), induct ancestors arbitrary: ptr)
  case Nil
  then show ?case
    by(auto)
next
  case (Cons a ancestors)
  then obtain x where x: "h ⊢ get_ancestors x →r a # ancestors"
    by(auto simp add: get_ancestors_def[of a] elim!: bind_returns_result_E2 split: option.splits)
  then have "x = a"
    by(auto simp add: get_ancestors_def[of x] elim!: bind_returns_result_E2 split: option.splits)
  then show ?case
    using Cons.hyps Cons.prems(2) get_ancestors_ptr_in_heap x
    by (metis assms(1) assms(2) assms(3) get_ancestors_obtains_children get_child_nodes_ptr_in_heap
          is_OK_returns_result_I)
qed


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 (insert assms(1) assms(5) assms(6), induct ptr' arbitrary: ancestors'
      rule: heap_wellformed_induct)
  case (step parent)
  then show ?case
  proof (cases "parent ≠ ptr" )
    case True

    then obtain children ancestor_child where "h ⊢ get_child_nodes parent →r children"
          and "ancestor_child ∈ set children" and "cast ancestor_child ∈ set ancestors"
    using assms(1) assms(2) assms(3) assms(4) get_ancestors_obtains_children step.prems(1) by blast
  then have "h ⊢ get_parent ancestor_child →r Some parent"
    using assms(1) assms(2) assms(3) child_parent_dual by blast
  then have "h ⊢ get_ancestors (cast ancestor_child) →r cast ancestor_child # ancestors'"
    apply(simp add: get_ancestors_def)
    using ‹h ⊢ get_ancestors parent →r ancestors'› get_parent_ptr_in_heap
    by(auto simp add: check_in_heap_def is_OK_returns_result_I intro!: bind_pure_returns_result_I)
  then show ?thesis
    using step(1) ‹h ⊢ get_child_nodes parent →r children› ‹ancestor_child ∈ set children›
      ‹cast ancestor_child ∈ set ancestors› ‹h ⊢ get_ancestors (cast ancestor_child) →r cast ancestor_child
# ancestors'›
    by fastforce
  next
    case False
    then show ?thesis
      by (metis append_Nil assms(4) returns_result_eq step.prems(2))
  qed
```

```
qed


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 -
  have "ptr' |∈| object_ptr_kinds h"
    by (metis assms(1) assms(2) assms(3) assms(4) assms(5) get_ancestors_obtains_children
              get_ancestors_ptr_in_heap get_child_nodes_ptr_in_heap is_OK_returns_result_I)
  then obtain ancestors' where ancestors': "h ⊢ get_ancestors ptr' →r ancestors'"
    by (meson assms(1) assms(2) assms(3) get_ancestors_ok is_OK_returns_result_E)
  then have "∃pre. ancestors = pre @ ancestors'"
    using get_ancestors_prefix assms by blast
  moreover have "ptr'' |∈| object_ptr_kinds h"
    by (metis assms(1) assms(2) assms(3) assms(4) assms(6) get_ancestors_obtains_children
              get_ancestors_ptr_in_heap get_child_nodes_ptr_in_heap is_OK_returns_result_I)
  then obtain ancestors'' where ancestors'': "h ⊢ get_ancestors ptr'' →r ancestors''"
    by (meson assms(1) assms(2) assms(3) get_ancestors_ok is_OK_returns_result_E)
  then have "∃pre. ancestors = pre @ ancestors''"
    using get_ancestors_prefix assms by blast
  ultimately show ?thesis
    using ancestors' ancestors''
    apply(auto simp add: get_root_node_def elim!: bind_returns_result_E2
        intro!: bind_pure_returns_result_I)[1]
     apply (metis (no_types, lifting) assms(1) get_ancestors_never_empty last_appendR
                                      returns_result_eq)
    by (metis assms(1) get_ancestors_never_empty last_appendR returns_result_eq)
qed

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
  assume 1: " h ⊢ get_root_node (cast_node_ptr2object_ptr child) →r root"
  show "h ⊢ get_root_node ptr →r root"
    using 1[unfolded get_root_node_def] assms
    apply(simp add: get_ancestors_def)
    apply(auto simp add: get_root_node_def dest: returns_result_eq elim!: bind_returns_result_E2
        intro!: bind_pure_returns_result_I split: option.splits)[1]
    using returns_result_eq apply fastforce
    using get_ancestors_ptr by fastforce
next
  assume 1: " h ⊢ get_root_node ptr →r root"
  show "h ⊢ get_root_node (cast_node_ptr2object_ptr child) →r root"
    apply(simp add: get_root_node_def)
    using assms 1
    apply(simp add: get_ancestors_def)
    apply(auto simp add: get_root_node_def dest: returns_result_eq elim!: bind_returns_result_E2
              intro!: bind_pure_returns_result_I split: option.splits)[1]
     apply (simp add: check_in_heap_def is_OK_returns_result_I)
    using get_ancestors_ptr get_parent_ptr_in_heap
    apply (simp add: is_OK_returns_result_I)
    by (meson list.distinct(1) list.set_cases local.get_ancestors_ptr)
qed

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 (insert assms(1) assms(4), induct ptr rule: heap_wellformed_induct_rev)
  case (step c)
  then show ?case
  proof (cases "cast_{object_ptr2node_ptr} c")
    case None
    then have "c = cast child"
      using step(2)
      by(auto simp add: get_root_node_def get_ancestors_def[of c] elim!: bind_returns_result_E2)
    then show ?thesis
      using None by auto
  next
    case (Some child_node)
    note s = this
    then obtain parent_opt where parent_opt: "h ⊢ get_parent child_node →_r parent_opt"
      by (metis (no_types, lifting) assms(2) assms(3) get_root_node_ptr_in_heap
                is_OK_returns_result_I local.get_parent_ok node_ptr_casts_commute
                node_ptr_kinds_commutes returns_result_select_result step.prems)
    then show ?thesis
    proof(induct parent_opt)
      case None
      then show ?case
        using Some get_root_node_no_parent returns_result_eq step.prems by fastforce
    next
      case (Some parent)
      then show ?case
        using step s
        apply(auto simp add: get_root_node_def get_ancestors_def[of c]
                    elim!: bind_returns_result_E2 split: option.splits list.splits)[1]
        using get_root_node_parent_same step.hyps step.prems by auto
    qed
  qed
qed

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"
  using assms
  apply(simp add: get_root_node_def get_ancestors_def)
  by(auto simp add: get_root_node_def dest: returns_result_eq elim!: bind_returns_result_E2
          intro!: bind_pure_returns_result_I split: option.splits)


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"
  using assms
  apply(auto simp add: get_root_node_def elim!: bind_returns_result_E2)[1]
  by (simp add: get_ancestors_never_empty get_ancestors_ptrs_in_heap)


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))"
  by (metis (no_types, lifting) assms(1) assms(2) assms(3) assms(4) get_root_node_same_no_parent
            l_heap_is_wellformed.parent_child_rel_child local.child_parent_dual local.get_child_nodes_ok

            local.known_ptrs_known_ptr local.l_heap_is_wellformed_axioms local.parent_child_rel_node_ptr
            local.parent_child_rel_parent_in_heap node_ptr_casts_commute3 option.simps(3) returns_result_eq

            returns_result_select_result)
```

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_not_node_same:*
"ptr |∈| object_ptr_kinds h ⟹ ¬is_node_ptr_kind ptr

```
                                    ⟹ h ⊢ get_root_node ptr →r ptr"
  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_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
  using instances
  by(simp add: l_get_root_node_wf_Core_DOM_def)
declare l_get_root_node_wf_Core_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"
  using known_ptrs_is_l_known_ptrs
  apply(auto simp add: l_get_ancestors_wf_def l_get_ancestors_wf_axioms_def)[1]
  using get_ancestors_never_empty apply blast
  using get_ancestors_ok apply blast
  using get_ancestors_reads apply blast
  using get_ancestors_ptrs_in_heap apply blast
  using get_ancestors_remains_not_in_ancestors apply blast
  using get_ancestors_also_parent apply blast
  using get_ancestors_obtains_children apply blast
  using get_ancestors_parent_child_rel apply blast
  using get_ancestors_parent_child_rel apply blast
  done

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"
  using known_ptrs_is_l_known_ptrs
  apply(auto simp add: l_get_root_node_wf_def l_get_root_node_wf_axioms_def)[1]
  using get_root_node_ok apply blast
  using get_root_node_ptr_in_heap apply blast
  using get_root_node_root_in_heap apply blast
  using get_ancestors_same_root_node apply(blast, blast)
  using get_root_node_same_no_parent apply blast
  using get_root_node_not_node_same apply blast
  using get_root_node_parent_same apply (blast, blast)
  done
```

6.3.5 to_tree_order

```
locale l_to_tree_order_wf_Core_DOM =
  l_to_tree_order_Core_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(insert assms(1) assms(4), induct rule: heap_wellformed_induct)
  case (step parent)
  then show ?case
    apply(auto simp add: to_tree_order_def[of parent] map_M_pure_I elim!: bind_is_OK_E3)[1]
    using get_child_nodes_ptr_in_heap by blast
qed
```

```
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 -
  obtain treeorders where treeorders: "h ⊢ map_M to_tree_order (map cast children) →r treeorders"
    using assms
    apply(auto simp add: to_tree_order_def elim!: bind_returns_result_E)[1]
    using pure_returns_heap_eq returns_result_eq by fastforce
  then have "node ∈ set (concat treeorders)"
    using assms[simplified to_tree_order_def]
    by(auto elim!: bind_returns_result_E4 dest: pure_returns_heap_eq)
  then obtain treeorder where "treeorder ∈ set treeorders"
      and node_in_treeorder: "node ∈ set treeorder"
    by auto
  then obtain child where "h ⊢ to_tree_order (cast_{node_ptr2object_ptr} child) →r treeorder"
      and  "child ∈ set children"
    using assms[simplified to_tree_order_def] treeorders
    by(auto elim!: map_M_pure_E2)
  then show ?thesis
    using node_in_treeorder returns_result_eq that by auto
qed


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(insert assms(1) assms(4) assms(5), induct ptr arbitrary: to rule: heap_wellformed_induct)
  case (step parent)
  have "parent |∈| object_ptr_kinds h"
    using assms(1) assms(2) assms(3) step.prems(1) to_tree_order_ptr_in_heap by blast
  then obtain children where children: "h ⊢ get_child_nodes parent →r children"
    by (meson assms(2) assms(3) get_child_nodes_ok is_OK_returns_result_E local.known_ptrs_known_ptr)
  then show ?case
  proof (cases "children = []")
    case True
    then have "to = [parent]"
      using step(2) children
      apply(auto simp add: to_tree_order_def[of parent] map_M_pure_I elim!: bind_returns_result_E2)[1]
      by (metis list.distinct(1) list.map_disc_iff list.set_cases map_M_pure_E2 returns_result_eq)
    then show ?thesis
      using ⟨parent |∈| object_ptr_kinds h⟩ step.prems(2) by auto
  next
    case False
    note f = this
    then show ?thesis
      using children step to_tree_order_either_ptr_or_in_children
    proof (cases "ptr' = parent")
      case True
      then show ?thesis
        using ⟨parent |∈| object_ptr_kinds h⟩ by blast
    next
      case False
    then show ?thesis
      using children step.hyps to_tree_order_either_ptr_or_in_children
      by (metis step.prems(1) step.prems(2))
    qed
```

```
  qed
qed

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(insert assms(1) assms(2), induct rule: heap_wellformed_induct)
  case (step parent)
  then show ?case
    using assms(3) type_wf
    apply(simp add: to_tree_order_def)
    apply(auto simp add: heap_is_wellformed_def intro!: map_M_ok_I bind_is_OK_pure_I map_M_pure_I)[1]
    using get_child_nodes_ok known_ptrs_known_ptr apply blast
    by (simp add: local.heap_is_wellformed_children_in_heap local.to_tree_order_def wellformed)
qed

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
  fix x
  assume a1: "x ∈ set nodes'"
  moreover obtain treeorders
          where treeorders: "h ⊢ map_M to_tree_order (map cast children) →r treeorders"
    using assms(2) assms(3)
    apply(auto simp add: to_tree_order_def elim!: bind_returns_result_E)[1]
    using pure_returns_heap_eq returns_result_eq by fastforce
  then have "nodes' ∈ set treeorders"
    using assms(4) assms(5)
    by(auto elim!: map_M_pure_E dest: returns_result_eq)
  moreover have "set (concat treeorders) ⊆ set nodes"
    using treeorders assms(2) assms(3)
    by(auto simp add: to_tree_order_def elim!: bind_returns_result_E4 dest: pure_returns_heap_eq)
  ultimately show "x ∈ set nodes"
    by auto
qed

lemma to_tree_order_ptr_in_result:
  assumes "h ⊢ to_tree_order ptr →r nodes"
  shows "ptr ∈ set nodes"
  using assms
  apply(simp add: to_tree_order_def)
  by(auto elim!: bind_returns_result_E2 intro!: map_M_pure_I bind_pure_I)

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 -
  have "∀nodes. h ⊢ to_tree_order ptr →r nodes ⟶ (∀node. node ∈ set nodes
      ⟶ (∀nodes'. h ⊢ to_tree_order node →r nodes' ⟶ set nodes' ⊆ set nodes))"
  proof(insert assms(1), induct ptr rule: heap_wellformed_induct)
```

```
    case (step parent)
    then show ?case
    proof safe
      fix nodes node nodes' x
      assume 1: "(⋀children child.
            h ⊢ get_child_nodes parent →r children ⟹
            child ∈ set children ⟹ ∀nodes. h ⊢ to_tree_order (cast child) →r nodes
           ⟶ (∀node. node ∈ set nodes ⟶ (∀nodes'. h ⊢ to_tree_order node →r nodes'
                                          ⟶ set nodes' ⊆ set nodes)))"
        and 2: "h ⊢ to_tree_order parent →r nodes"
        and 3: "node ∈ set nodes"
        and "h ⊢ to_tree_order node →r nodes'"
        and "x ∈ set nodes'"
      have h1: "(⋀children child nodes node nodes'.
            h ⊢ get_child_nodes parent →r children ⟹
            child ∈ set children ⟹ h ⊢ to_tree_order (cast child) →r nodes
            ⟶ (node ∈ set nodes ⟶ (h ⊢ to_tree_order node →r nodes' ⟶ set nodes' ⊆ set nodes)))"
        using 1
        by blast
      obtain children where children: "h ⊢ get_child_nodes parent →r children"
        using 2
        by(auto simp add: to_tree_order_def elim!: bind_returns_result_E)
      then have "set nodes' ⊆ set nodes"
      proof (cases "children = []")
        case True
        then show ?thesis
          by (metis "2" "3" ‹h ⊢ to_tree_order node →r nodes'› children empty_iff list.set(1)
                    subsetI to_tree_order_either_ptr_or_in_children)
      next
        case False
        then show ?thesis
        proof (cases "node = parent")
          case True
          then show ?thesis
            using "2" ‹h ⊢ to_tree_order node →r nodes'› returns_result_eq by fastforce
        next
          case False
          then obtain child nodes_of_child where
            "child ∈ set children" and
            "h ⊢ to_tree_order (cast child) →r nodes_of_child" and
            "node ∈ set nodes_of_child"
            using 2[simplified to_tree_order_def] 3
                  to_tree_order_either_ptr_or_in_children[where node=node and ptr=parent] children
            apply(auto elim!: bind_returns_result_E2 intro: map_M_pure_I)[1]
            using is_OK_returns_result_E 2 a_all_ptrs_in_heap_def assms(1) heap_is_wellformed_def
            using "3" by blast
          then have "set nodes' ⊆ set nodes_of_child"
            using h1
            using ‹h ⊢ to_tree_order node →r nodes'› children by blast
          moreover have "set nodes_of_child ⊆ set nodes"
            using "2" ‹child ∈ set children› ‹h ⊢ to_tree_order (cast child) →r nodes_of_child›
                  assms children to_tree_order_child_subset by auto
          ultimately show ?thesis
            by blast
        qed
      qed
      then show "x ∈ set nodes"
        using ‹x ∈ set nodes'› by blast
    qed
  qed
  then show ?thesis
    using assms by blast
qed
```

```
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 -
  obtain nodes' where nodes': "h ⊢ to_tree_order parent →r nodes'"
    using assms to_tree_order_ok get_parent_parent_in_heap
    by (meson get_parent_parent_in_heap is_OK_returns_result_E)

  then have "set nodes' ⊆ set nodes"
    using to_tree_order_subset assms
    by blast
  moreover obtain children where
    children: "h ⊢ get_child_nodes parent →r children" and
    child: "child ∈ set children"
    using assms get_parent_child_dual by blast
  then obtain child_to where child_to: "h ⊢ to_tree_order (cast_{node_ptr2object_ptr} child) →r child_to"
    by (meson assms(1) assms(2) assms(3) assms(5) is_OK_returns_result_E is_OK_returns_result_I
              get_parent_ptr_in_heap node_ptr_kinds_commutes to_tree_order_ok)
  then have "cast child ∈ set child_to"
    apply(simp add: to_tree_order_def)
    by(auto elim!: bind_returns_result_E2 map_M_pure_E
           dest!: bind_returns_result_E3[rotated, OF children, rotated]  intro!: map_M_pure_I)

  have "cast child ∈ set nodes'"
    using nodes' child
    apply(simp add: to_tree_order_def)
    apply(auto elim!: bind_returns_result_E2 map_M_pure_E
              dest!: bind_returns_result_E3[rotated, OF children, rotated]  intro!: map_M_pure_I)[1]
    using child_to ‹cast child ∈ set child_to› returns_result_eq by fastforce
  ultimately show ?thesis
    by auto
qed

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(insert assms(1) assms(4) assms(6) assms(8), induct ptr arbitrary: nodes
            rule: heap_wellformed_induct)
  case (step p)
  have "p |∈| object_ptr_kinds h"
    using ‹h ⊢ to_tree_order p →r nodes› to_tree_order_ptr_in_heap
    using assms(1) assms(2) assms(3) by blast
  then obtain children where children: "h ⊢ get_child_nodes p →r children"
    by (meson assms(2) assms(3) get_child_nodes_ok is_OK_returns_result_E local.known_ptrs_known_ptr)
  then show ?case
  proof (cases "children = []")
    case True
    then show ?thesis
      using step(2) step(3) step(4) children
      by(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
             dest!: bind_returns_result_E3[rotated, OF children, rotated])
  next
    case False
    then obtain c child_to where
```

```
      child: "c ∈ set children" and
      child_to: "h ⊢ to_tree_order (cast_node_ptr2object_ptr c) →_r child_to" and
      "cast child ∈ set child_to"
      using step(2) children
      apply(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
                dest!: bind_returns_result_E3[rotated, OF children, rotated])[1]
      by (metis (full_types) assms(1) assms(2) assms(3) get_parent_ptr_in_heap
                is_OK_returns_result_I l_get_parent_wf_Core_DOM.child_parent_dual
                l_get_parent_wf_Core_DOM_axioms node_ptr_kinds_commutes
                returns_result_select_result step.prems(1) step.prems(2) step.prems(3)
                to_tree_order_either_ptr_or_in_children to_tree_order_ok)
    then have "set child_to ⊆ set nodes"
      using assms(1) child children step.prems(1) to_tree_order_child_subset by auto

    show ?thesis
    proof (cases "c = child")
      case True
      then have "parent = p"
        using step(3) children child assms(5) assms(7)
        by (meson assms(1) assms(2) assms(3) child_parent_dual option.inject returns_result_eq)

      then show ?thesis
        using step.prems(1) to_tree_order_ptr_in_result by blast
    next
      case False
      then show ?thesis
      using step(1)[OF children child child_to] step(3) step(4)
      using ⟨set child_to ⊆ set nodes⟩
      using ⟨cast_node_ptr2object_ptr child ∈ set child_to⟩ by auto
  qed
  qed
qed

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(insert assms(1) assms(4) assms(5) assms(6), induct ptr arbitrary: nodes
            rule: heap_wellformed_induct)
  case (step p)
  have "p |∈| object_ptr_kinds h"
    using ⟨h ⊢ to_tree_order p →_r nodes⟩ to_tree_order_ptr_in_heap
    using assms(1) assms(2) assms(3) by blast
  then obtain children where children: "h ⊢ get_child_nodes p →_r children"
    by (meson assms(2) assms(3) get_child_nodes_ok is_OK_returns_result_E local.known_ptrs_known_ptr)
  then show ?case
  proof (cases "children = []")
    case True
    then show ?thesis
      using step(2) step(3) step(4) children
      by(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
              dest!: bind_returns_result_E3[rotated, OF children, rotated])[1]
  next
    case False
    then obtain c child_to where
      child: "c ∈ set children" and
      child_to: "h ⊢ to_tree_order (cast_node_ptr2object_ptr c) →_r child_to" and
      "ptr' ∈ set child_to"
      using step(2) children
      apply(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
                dest!: bind_returns_result_E3[rotated, OF children, rotated])[1]
```

```
    using step.prems(1) step.prems(2) step.prems(3) to_tree_order_either_ptr_or_in_children by blast
    then have "set child_to ⊆ set nodes"
      using assms(1) child children step.prems(1) to_tree_order_child_subset by auto

    show ?thesis
    proof (cases "cast c = ptr")
      case True
      then show ?thesis
        using step ⟨ptr' ∈ set child_to⟩ assms(5) child child_to children by blast
    next
      case False
      then show ?thesis
        using ⟨ptr' ∈ set child_to⟩ child child_to children is_node_ptr_kind_cast step.hyps by blast
    qed
  qed
qed

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 -
  assume 1: "(⋀parent. h ⊢ get_parent child →r Some parent ⟹ parent ∈ set nodes ⟹ thesis)"
  show thesis
  proof(insert assms(1) assms(4) assms(5) assms(6) 1, induct ptr arbitrary: nodes
              rule: heap_wellformed_induct)
    case (step p)
    have "p |∈| object_ptr_kinds h"
      using ⟨h ⊢ to_tree_order p →r nodes⟩ to_tree_order_ptr_in_heap
      using assms(1) assms(2) assms(3) by blast
    then obtain children where children: "h ⊢ get_child_nodes p →r children"
      by (meson assms(2) assms(3) get_child_nodes_ok is_OK_returns_result_E local.known_ptrs_known_ptr)
    then show ?case
    proof (cases "children = []")
      case True
      then show ?thesis
        using step(2) step(3) step(4) children
        by(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
              dest!: bind_returns_result_E3[rotated, OF children, rotated])
    next
      case False
      then obtain c child_to where
        child: "c ∈ set children" and
        child_to: "h ⊢ to_tree_order (cast_{node_ptr2object_ptr} c) →r child_to" and
        "cast child ∈ set child_to"
        using step(2) children
        apply(auto simp add: to_tree_order_def[of p] map_M_pure_I  elim!: bind_returns_result_E2
                dest!: bind_returns_result_E3[rotated, OF children, rotated])[1]
        using step.prems(1) step.prems(2) step.prems(3) to_tree_order_either_ptr_or_in_children
        by blast
      then have "set child_to ⊆ set nodes"
        using assms(1) child children step.prems(1) to_tree_order_child_subset by auto

      have "cast child |∈| object_ptr_kinds h"
        using assms(1) assms(2) assms(3) assms(4) assms(6) to_tree_order_ptrs_in_heap by blast
      then obtain parent_opt where parent_opt: "h ⊢ get_parent child →r parent_opt"
        by (meson assms(2) assms(3) is_OK_returns_result_E get_parent_ok node_ptr_kinds_commutes)
      then show ?thesis
      proof (induct parent_opt)
        case None
        then show ?case
```

```
      by (metis ⟨cast_node_ptr2object_ptr child ∈ set child_to⟩ assms(1) assms(2) assms(3)
                cast_node_ptr2object_ptr_inject child child_parent_dual child_to children
                option.distinct(1) returns_result_eq step.hyps)
    next
      case (Some option)
      then show ?case
        by (meson assms(1) assms(2) assms(3) get_parent_child_dual step.prems(1) step.prems(2)
            step.prems(3) step.prems(4) to_tree_order_child)
    qed
  qed
qed

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
  assume 3: "(ptr, child) ∈ (parent_child_rel h)*"
  show "child ∈ set to"
  proof (insert 3, induct child rule: heap_wellformed_induct_rev[OF assms(1)])
    case (1 child)
    then show ?case
    proof (cases "ptr = child")
      case True
      then show ?thesis
        using assms(4)
        apply(simp add: to_tree_order_def)
        by(auto simp add: map_M_pure_I elim!: bind_returns_result_E2)
    next
      case False
      obtain child_parent where
        "(ptr, child_parent) ∈ (parent_child_rel h)*" and
        "(child_parent, child) ∈ (parent_child_rel h)"
        using ⟨ptr ≠ child⟩
        by (metis "1.prems" rtranclE)
      obtain child_node where child_node: "cast_node_ptr2object_ptr child_node = child"
        using ⟨(child_parent, child) ∈ parent_child_rel h⟩ node_ptr_casts_commute3
              parent_child_rel_node_ptr
        by blast
      then have "h ⊢ get_parent child_node →_r Some child_parent"
        using ⟨(child_parent, child) ∈ (parent_child_rel h)⟩
        by (meson assms(1) assms(2) assms(3) is_OK_returns_result_E l_get_parent_wf.child_parent_dual
                  l_heap_is_wellformed.parent_child_rel_child local.get_child_nodes_ok
                  local.known_ptrs_known_ptr local.l_get_parent_wf_axioms
                  local.l_heap_is_wellformed_axioms local.parent_child_rel_parent_in_heap)
      then show ?thesis
        using 1(1) child_node ⟨(ptr, child_parent) ∈ (parent_child_rel h)*⟩
        using assms(1) assms(2) assms(3) assms(4) to_tree_order_parent by blast
    qed
  qed
next
  assume "child ∈ set to"
  then show "(ptr, child) ∈ (parent_child_rel h)*"
  proof (induct child rule: heap_wellformed_induct_rev[OF assms(1)])
    case (1 child)
    then show ?case
    proof (cases "ptr = child")
      case True
      then show ?thesis
        by simp
    next
      case False
```

```
      then have "∃parent. (parent, child) ∈ (parent_child_rel h)"
        using 1(2) assms(4) to_tree_order_child2[OF assms(1) assms(2) assms(3) assms(4)]
              to_tree_order_node_ptrs
        by (metis assms(1) assms(2) assms(3) node_ptr_casts_commute3 parent_child_rel_parent)
      then obtain child_node where child_node: "cast_{node_ptr2object_ptr} child_node = child"
        using node_ptr_casts_commute3 parent_child_rel_node_ptr by blast
      then obtain child_parent where child_parent: "h ⊢ get_parent child_node →_r Some child_parent"
        using ‹∃parent. (parent, child) ∈ (parent_child_rel h)›
        by (metis "1.prems" False assms(1) assms(2) assms(3) assms(4) to_tree_order_child2)
      then have "(child_parent, child) ∈ (parent_child_rel h)"
        using assms(1) child_node parent_child_rel_parent by blast
      moreover have "child_parent ∈ set to"
        by (metis "1.prems" False assms(1) assms(2) assms(3) assms(4) child_node child_parent
                  get_parent_child_dual to_tree_order_child)
      then have "(ptr, child_parent) ∈ (parent_child_rel h)^*"
        using 1 child_node child_parent by blast
      ultimately show ?thesis
        by auto
    qed
  qed
qed
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
  using instances
  apply(simp add: l_to_tree_order_wf_{Core_DOM}_def)
  done
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"
  using instances
  apply(auto simp add: l_to_tree_order_wf_def l_to_tree_order_wf_axioms_def)[1]
  using to_tree_order_ok
         apply blast
  using to_tree_order_ptrs_in_heap
        apply blast
  using to_tree_order_parent_child_rel
       apply(blast, blast)
  using to_tree_order_child2
     apply blast
  using to_tree_order_node_ptrs
    apply blast
  using to_tree_order_child
   apply blast
  using to_tree_order_ptr_in_result
  apply blast
  using to_tree_order_parent
 apply blast
  using to_tree_order_subset
  apply blast
  done
```

get_root_node

```
locale l_to_tree_order_wf_get_root_node_wf_Core_DOM =
  l_get_ancestors
  + l_get_ancestors_wf
  + l_get_root_node
  + l_get_root_node_wf
  + l_to_tree_order_wf
  + l_get_parent
  + l_get_parent_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 -
  obtain ancestors' where ancestors': "h ⊢ get_ancestors ptr' →r ancestors'"
    by (meson assms(1) assms(2) assms(3) assms(4) assms(5) get_ancestors_ok is_OK_returns_result_E
              to_tree_order_ptrs_in_heap )
  moreover have "ptr ∈ set ancestors'"
    using ‹h ⊢ get_ancestors ptr' →r ancestors'›
    using assms(1) assms(2) assms(3) assms(4) assms(5) get_ancestors_parent_child_rel
          to_tree_order_parent_child_rel by blast

  ultimately have "h ⊢ get_root_node ptr →r root_ptr"
    using ‹h ⊢ get_root_node ptr' →r root_ptr›
    using assms(1) assms(2) assms(3) get_ancestors_ptr get_ancestors_same_root_node by blast

  obtain ancestors'' where ancestors'': "h ⊢ get_ancestors ptr'' →r ancestors''"
```

```
    by (meson assms(1) assms(2) assms(3) assms(4) assms(7) get_ancestors_ok is_OK_returns_result_E
          to_tree_order_ptrs_in_heap)
  moreover have "ptr ∈ set ancestors''"
    using ⟨h ⊢ get_ancestors ptr'' →r ancestors''⟩
    using assms(1) assms(2) assms(3) assms(4) assms(7) get_ancestors_parent_child_rel
      to_tree_order_parent_child_rel by blast
  ultimately show ?thesis
    using ⟨h ⊢ get_root_node ptr →r root_ptr⟩ assms(1) assms(2) assms(3) get_ancestors_ptr
      get_ancestors_same_root_node by blast
qed

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 (insert assms(1) assms(6), induct ptr' rule: heap_wellformed_induct_rev)
  case (step child)
  then show ?case
  proof (cases "h ⊢ get_root_node child →r child")
    case True
    then have "child = root_ptr"
      using  assms(1) assms(2) assms(3) assms(5) step.prems
      by (metis (no_types, lifting) get_root_node_same_no_parent node_ptr_casts_commute3
          option.simps(3) returns_result_eq to_tree_order_child2 to_tree_order_node_ptrs)
    then show ?thesis
      using True by blast
  next
    case False
    then obtain child_node parent where "cast child_node = child"
                and "h ⊢ get_parent child_node →r Some parent"
      by (metis assms(1) assms(2) assms(3) assms(4) assms(5) local.get_root_node_no_parent
          local.get_root_node_not_node_same local.get_root_node_same_no_parent
          local.to_tree_order_child2 local.to_tree_order_ptrs_in_heap node_ptr_casts_commute3
          step.prems)
    then show ?thesis
    proof (cases "child = root_ptr")
      case True
      then have "h ⊢ get_root_node root_ptr →r root_ptr"
        using assms(4)
        using ⟨cast_node_ptr2object_ptr child_node = child⟩ assms(1) assms(2) assms(3)
          get_root_node_no_parent get_root_node_same_no_parent
        by blast
      then show ?thesis
        using step assms(4)
        using True by blast
    next
      case False
      then have "parent ∈ set to"
        using assms(5) step(2) to_tree_order_child ⟨h ⊢ get_parent child_node →r Some parent⟩
          ⟨cast child_node = child⟩
        by (metis False assms(1) assms(2) assms(3) get_parent_child_dual)
      then show ?thesis
        using ⟨cast_node_ptr2object_ptr child_node = child⟩ ⟨h ⊢ get_parent child_node →r Some parent⟩
          get_root_node_parent_same
        using step.hyps by blast
    qed

  qed
qed
end
```

```
interpretation i_to_tree_order_wf_get_root_node_wf?: l_to_tree_order_wf_get_root_node_wf_Core_DOM
               get_ancestors get_ancestors_locs heap_is_wellformed parent_child_rel known_ptr
               known_ptrs type_wf get_child_nodes get_child_nodes_locs get_parent get_parent_locs
               get_root_node get_root_node_locs to_tree_order
  using instances
  by(simp add: l_to_tree_order_wf_get_root_node_wf_Core_DOM_def)

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"
  using instances
  apply(auto simp add: l_to_tree_order_wf_get_root_node_wf_def
      l_to_tree_order_wf_get_root_node_wf_axioms_def)[1]
  using to_tree_order_get_root_node apply blast
  using to_tree_order_same_root apply blast
  done
```

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
  + l_get_ancestors
  + l_get_ancestors_wf
  + l_get_parent
  + l_get_parent_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 -
  have 2: "node_ptr |∈| node_ptr_kinds h"
    using assms heap_is_wellformed_disc_nodes_in_heap
    by blast
  have 3: "document_ptr |∈| document_ptr_kinds h"
    using assms(2) get_disconnected_nodes_ptr_in_heap by blast
  have 0:
  "∃!document_ptr∈set |h ⊢ document_ptr_kinds_M|r. node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|r"
    by (metis (no_types, lifting) "3" DocumentMonad.ptr_kinds_ptr_kinds_M assms(1) assms(2) assms(3)
              disjoint_iff_not_equal l_heap_is_wellformed.heap_is_wellformed_one_disc_parent
              local.get_disconnected_nodes_ok local.l_heap_is_wellformed_axioms
              returns_result_select_result select_result_I2 type_wf)

  have "h ⊢ get_parent node_ptr →r None"
```

```
  using heap_is_wellformed_children_disc_nodes_different child_parent_dual assms
  using "2" disjoint_iff_not_equal local.get_parent_child_dual local.get_parent_ok
        returns_result_select_result split_option_ex
  by (metis (no_types, lifting))

then have 4: "h ⊢ get_root_node (cast node_ptr) →r cast node_ptr"
  using 2 get_root_node_no_parent
  by blast
obtain document_ptrs where document_ptrs: "h ⊢ document_ptr_kinds_M →r document_ptrs"
  by simp

then
have "h ⊢ ok (filter_M (λdocument_ptr. do {
        disconnected_nodes ← get_disconnected_nodes document_ptr;
        return (((cast_node_ptr2object_ptr node_ptr)) ∈ cast ` set disconnected_nodes)
      }) document_ptrs)"
  using assms(1) get_disconnected_nodes_ok  type_wf unfolding heap_is_wellformed_def
  by(auto intro!: bind_is_OK_I2 filter_M_is_OK_I bind_pure_I)
then obtain candidates where
  candidates: "h ⊢ filter_M (λdocument_ptr. do {
        disconnected_nodes ← get_disconnected_nodes document_ptr;
        return (((cast_node_ptr2object_ptr node_ptr)) ∈ cast ` set disconnected_nodes)
      }) document_ptrs →r candidates"
  by auto


have eq: "⋀document_ptr. document_ptr |∈| document_ptr_kinds h
         ⟹ node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|r ⟷ |h ⊢ do {
        disconnected_nodes ← get_disconnected_nodes document_ptr;
        return (((cast_node_ptr2object_ptr node_ptr)) ∈ cast ` set disconnected_nodes)
      }|r"
  apply(auto dest!: get_disconnected_nodes_ok[OF type_wf]
            intro!: select_result_I[where P=id, simplified] elim!: bind_returns_result_E2)[1]
  apply(drule select_result_E[where P=id, simplified])
  by(auto elim!: bind_returns_result_E2)

have filter: "filter (λdocument_ptr. |h ⊢ do {
        disconnected_nodes ← get_disconnected_nodes document_ptr;
        return (cast_node_ptr2object_ptr node_ptr ∈ cast ` set disconnected_nodes)
      }|r) document_ptrs = [document_ptr]"
  apply(rule filter_ex1)
  using 0 document_ptrs apply(simp)[1]
  using eq
  using local.get_disconnected_nodes_ok apply auto[1]
  using assms(2) assms(3)
    apply(auto intro!: intro!: select_result_I[where P=id, simplified]
              elim!: bind_returns_result_E2)[1]
  using returns_result_eq apply fastforce
  using document_ptrs  3 apply(simp)
  using document_ptrs
  by simp
have "h ⊢ filter_M (λdocument_ptr. do {
        disconnected_nodes ← get_disconnected_nodes document_ptr;
        return (((cast_node_ptr2object_ptr node_ptr)) ∈ cast ` set disconnected_nodes)
      }) document_ptrs →r [document_ptr]"
  apply(rule filter_M_filter2)
  using get_disconnected_nodes_ok document_ptrs 3  assms(1)  type_wf filter
  unfolding heap_is_wellformed_def
  by(auto intro: bind_pure_I bind_is_OK_I2)

with 4 document_ptrs have "h ⊢ a_get_owner_document_node_ptr node_ptr () →r document_ptr"
  by(auto simp add: a_get_owner_document_node_ptr_def
          intro!:  bind_pure_returns_result_I filter_M_pure_I bind_pure_I
```

```
          split: option.splits)[1]
  moreover have "known_ptr (cast_{node_ptr2object_ptr} node_ptr)"
    using "4" assms(1)  known_ptrs type_wf known_ptrs_known_ptr "2" node_ptr_kinds_commutes by blast
  ultimately show ?thesis
    using 2
    apply(auto simp add: known_ptr_impl get_owner_document_def a_get_owner_document_tups_def)[1]
    apply(split invoke_splits, (rule conjI | rule impI)+)+
    apply(drule(1) known_ptr_not_document_ptr[folded known_ptr_impl])
    apply(drule(1)  known_ptr_not_character_data_ptr)
       apply(drule(1)  known_ptr_not_element_ptr)
       apply(simp add: NodeClass.known_ptr_defs)
    by(auto split: option.splits intro!: bind_pure_returns_result_I)
qed

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 -
  have 2: "cast node_ptr |∈| object_ptr_kinds h"
    using assms(3) get_owner_document_ptr_in_heap by fast
  then have 3: "h ⊢ get_root_node (cast node_ptr) →_r cast node_ptr"
    using assms(2) local.get_root_node_no_parent by blast

  have "¬(∃parent_ptr. parent_ptr |∈| object_ptr_kinds h ∧ node_ptr ∈ set |h ⊢ get_child_nodes parent_ptr|_r)"
    apply(auto)[1]
    using assms(2) child_parent_dual[OF assms(1)] type_wf
          assms(1) assms(5) get_child_nodes_ok known_ptrs_known_ptr option.simps(3)
          returns_result_eq returns_result_select_result
    by (metis (no_types, hide_lams))
  moreover have "node_ptr |∈| node_ptr_kinds h"
    using assms(2) get_parent_ptr_in_heap by blast
  thm heap_is_wellformed_children_disc_nodes_different
  ultimately
  have 0: "∃document_ptr∈set |h ⊢ document_ptr_kinds_M|_r. node_ptr ∈ set |h ⊢ get_disconnected_nodes
document_ptr|_r"
    by (metis DocumentMonad.ptr_kinds_ptr_kinds_M assms(1) finite_set_in heap_is_wellformed_children_disc_nodes)
  then  obtain document_ptr where
    document_ptr: "document_ptr∈set |h ⊢ document_ptr_kinds_M|_r" and
    node_ptr_in_disc_nodes: "node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|_r"
    by auto
  then show ?thesis
    using get_owner_document_disconnected_nodes known_ptrs type_wf assms
    using DocumentMonad.ptr_kinds_ptr_kinds_M assms(1) assms(3) assms(4) get_disconnected_nodes_ok
          returns_result_select_result select_result_I2
    by (metis (no_types, hide_lams) )
qed
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"

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_root_node get_root_node_locs get_parent get_parent_locs get_ancestors get_ancestors_locs
  get_owner_document
  using known_ptrs_is_l_known_ptrs
  using heap_is_wellformed_is_l_heap_is_wellformed
  using get_root_node_is_l_get_root_node
  using get_ancestors_is_l_get_ancestors
  using get_ancestors_wf_is_l_get_ancestors_wf
  using get_parent_is_l_get_parent
  using get_ancestors_wf_is_l_get_ancestors_wf
  using get_parent_wf_is_l_get_parent_wf
  using l_get_owner_document_Core_DOM_axioms
  by(simp add: l_get_owner_document_wf_Core_DOM_def)



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"
  using known_ptrs_is_l_known_ptrs
  apply(simp add: l_get_owner_document_wf_def l_get_owner_document_wf_axioms_def)
  using get_owner_document_disconnected_nodes in_disconnected_nodes_no_parent
  by fast
```

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
  apply(rule preserves_wellformedness_writes_needed[OF assms set_attribute_writes])
  using  set_attribute_get_child_nodes
  apply(fast)
  using set_attribute_get_disconnected_nodes apply(fast)
  by(auto simp add: all_args_def set_attribute_locs_def)
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 -
  obtain children where children: "h ⊢ get_child_nodes ptr →_r children"
    using remove_child remove_child_def by auto
  then have "child ∈ set children"
    using remove_child remove_child_def
    by(auto elim!: bind_returns_heap_E dest: returns_result_eq split: if_splits)
  then have h1: "⋀other_ptr other_children. other_ptr ≠ ptr
              ⟹ h ⊢ get_child_nodes other_ptr →_r other_children ⟹ child ∉ set other_children"
    using assms(1) known_ptrs  type_wf child_parent_dual
    by (meson child_parent_dual children option.inject returns_result_eq)

  have known_ptr: "known_ptr ptr"
    using known_ptrs
    by (meson is_OK_returns_heap_I l_known_ptrs.known_ptrs_known_ptr l_known_ptrs_axioms
          remove_child remove_child_ptr_in_heap)

  obtain owner_document disc_nodes h' where
    owner_document: "h ⊢ get_owner_document (cast child) →_r owner_document" and
    disc_nodes: "h ⊢  get_disconnected_nodes owner_document →_r disc_nodes" and
    h': "h ⊢  set_disconnected_nodes owner_document (child # disc_nodes) →_h h'" and
    h2: "h' ⊢ set_child_nodes ptr (remove1 child children) →_h h2"
    using assms children unfolding remove_child_def
    apply(auto split: if_splits elim!: bind_returns_heap_E)[1]
    by (metis (full_types) get_child_nodes_pure get_disconnected_nodes_pure
       get_owner_document_pure pure_returns_heap_eq returns_result_eq)

  have "object_ptr_kinds h = object_ptr_kinds h2"
    using remove_child_writes remove_child unfolding remove_child_locs_def
    apply(rule writes_small_big)
    using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
    by(auto simp add: reflp_def transp_def)
  then have "|h ⊢ object_ptr_kinds_M|_r = |h2 ⊢ object_ptr_kinds_M|_r"
    unfolding object_ptr_kinds_M_defs by simp

  have "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'",
                           OF set_disconnected_nodes_writes h']
    using set_disconnected_nodes_types_preserved type_wf
    by(auto simp add: reflp_def transp_def)
  have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'",
                           OF remove_child_writes remove_child]  unfolding remove_child_locs_def
    using set_disconnected_nodes_types_preserved set_child_nodes_types_preserved type_wf
    apply(auto simp add: reflp_def transp_def)[1]
    by blast
  then obtain children' where children': "h2 ⊢ get_child_nodes ptr →_r children'"
    using h2 set_child_nodes_get_child_nodes known_ptr
    by (metis ‹object_ptr_kinds h = object_ptr_kinds h2› children get_child_nodes_ok
              get_child_nodes_ptr_in_heap is_OK_returns_result_E is_OK_returns_result_I)

  have "child ∉ set children'"
    by (metis (mono_tags, lifting) ‹type_wf h'› children children' distinct_remove1_removeAll h2
              known_ptr local.heap_is_wellformed_children_distinct
              local.set_child_nodes_get_child_nodes member_remove remove_code(1) select_result_I2
              wellformed)
```

```
  moreover have "⋀other_ptr other_children. other_ptr ≠ ptr
                ⟹ h' ⊢ get_child_nodes other_ptr →r other_children ⟹ child ∉ set other_children"
  proof -
    fix other_ptr other_children
    assume a1: "other_ptr ≠ ptr"  and a3: "h' ⊢ get_child_nodes other_ptr →r other_children"
    have "h ⊢ get_child_nodes other_ptr →r other_children"
      using get_child_nodes_reads set_disconnected_nodes_writes h' a3
      apply(rule reads_writes_separate_backwards)
      using set_disconnected_nodes_get_child_nodes by fast
    show "child ∉ set other_children"
      using ‹h ⊢ get_child_nodes other_ptr →r other_children› a1 h1 by blast
  qed
  then have "⋀other_ptr other_children. other_ptr ≠ ptr
            ⟹  h2 ⊢ get_child_nodes other_ptr →r other_children ⟹ child ∉ set other_children"
  proof -
    fix other_ptr other_children
    assume a1: "other_ptr ≠ ptr" and a3: "h2 ⊢ get_child_nodes other_ptr →r other_children"
    have "h' ⊢ get_child_nodes other_ptr →r other_children"
      using get_child_nodes_reads set_child_nodes_writes h2 a3
      apply(rule reads_writes_separate_backwards)
      using set_disconnected_nodes_get_child_nodes a1 set_child_nodes_get_child_nodes_different_pointers

      by metis
    then show "child ∉ set other_children"
      using ‹⋀other_ptr other_children. ⟦other_ptr ≠ ptr; h' ⊢ get_child_nodes other_ptr →r other_children⟧

            ⟹ child ∉ set other_children› a1 by blast
  qed
  ultimately have ha: "⋀other_ptr other_children. h2 ⊢ get_child_nodes other_ptr →r other_children
                     ⟹ child ∉ set other_children"
    by (metis (full_types) children' returns_result_eq)
  moreover obtain ptrs where ptrs: "h2 ⊢ object_ptr_kinds_M →r ptrs"
    by (simp add: object_ptr_kinds_M_defs)
  moreover have "⋀ptr. ptr ∈ set ptrs ⟹ h2 ⊢ ok (get_child_nodes ptr)"
    using ‹type_wf h2› ptrs get_child_nodes_ok known_ptr
    using ‹object_ptr_kinds h = object_ptr_kinds h2› known_ptrs local.known_ptrs_known_ptr by auto
  ultimately show "h2 ⊢ get_parent child →r None"
    apply(auto simp add: get_parent_def intro!: bind_pure_returns_result_I filter_M_pure_I bind_pure_I)[1]
  proof -
    have "cast_node_ptr2object_ptr child |∈| object_ptr_kinds h"
      using get_owner_document_ptr_in_heap owner_document by blast
    then show "h2 ⊢ check_in_heap (cast_node_ptr2object_ptr child) →r ()"
      by (simp add: ‹object_ptr_kinds h = object_ptr_kinds h2› check_in_heap_def)
  next
    show "(⋀other_ptr other_children. h2 ⊢ get_child_nodes other_ptr →r other_children
         ⟹ child ∉ set other_children) ⟹
    ptrs = sorted_list_of_set (fset (object_ptr_kinds h2)) ⟹
    (⋀ptr. ptr |∈| object_ptr_kinds h2 ⟹ h2 ⊢ ok get_child_nodes ptr) ⟹
    h2 ⊢ filter_M (λptr. Heap_Error_Monad.bind (get_child_nodes ptr)
       (λchildren. return (child ∈ set children))) (sorted_list_of_set (fset (object_ptr_kinds h2))) →r
[]"
      by(auto intro!: filter_M_empty_I bind_pure_I)
  qed
qed
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 (standard, safe)

  obtain owner_document children_h h2 disconnected_nodes_h where
    owner_document: "h ⊢ get_owner_document (cast_node_ptr2object_ptr child) →_r owner_document" and
    children_h: "h ⊢ get_child_nodes ptr →_r children_h" and
    child_in_children_h: "child ∈ set children_h" and
    disconnected_nodes_h: "h ⊢  get_disconnected_nodes owner_document →_r disconnected_nodes_h" and
    h2: "h ⊢  set_disconnected_nodes owner_document (child # disconnected_nodes_h) →_h h2" and
    h': "h2 ⊢ set_child_nodes ptr (remove1 child children_h) →_h h'"
    using assms(2)
    apply(auto simp add: remove_child_def elim!: bind_returns_heap_E
              dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                     pure_returns_heap_eq[rotated, OF get_child_nodes_pure]
              split: if_splits)[1]
    using pure_returns_heap_eq by fastforce
  have object_ptr_kinds_eq3: "object_ptr_kinds h = object_ptr_kinds h'"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_eq: "⋀ptrs. h ⊢ object_ptr_kinds_M →_r ptrs = h' ⊢ object_ptr_kinds_M →_r
ptrs"
    unfolding object_ptr_kinds_M_defs by simp
  then have object_ptr_kinds_eq2: "|h ⊢ object_ptr_kinds_M|_r = |h' ⊢ object_ptr_kinds_M|_r"
    using select_result_eq by force
  then have node_ptr_kinds_eq2: "|h ⊢ node_ptr_kinds_M|_r = |h' ⊢ node_ptr_kinds_M|_r"
    using node_ptr_kinds_M_eq by auto
  then have node_ptr_kinds_eq3: "node_ptr_kinds h = node_ptr_kinds h'"
    using node_ptr_kinds_M_eq by auto
  have document_ptr_kinds_eq2: "|h ⊢ document_ptr_kinds_M|_r = |h' ⊢ document_ptr_kinds_M|_r"
    using object_ptr_kinds_eq2 document_ptr_kinds_M_eq by auto
  then have document_ptr_kinds_eq3: "document_ptr_kinds h = document_ptr_kinds h'"
    using document_ptr_kinds_M_eq by auto
  have children_eq:
    "⋀ptr' children. ptr ≠ ptr' ⟹ h ⊢ get_child_nodes ptr' →_r children = h' ⊢ get_child_nodes ptr'
→_r children"
    apply(rule reads_writes_preserved[OF get_child_nodes_reads remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_disconnected_nodes_get_child_nodes set_child_nodes_get_child_nodes_different_pointers
    by fast
  then have children_eq2:
    "⋀ptr' children. ptr ≠ ptr' ⟹ |h ⊢ get_child_nodes ptr'|_r = |h' ⊢ get_child_nodes ptr'|_r"
    using select_result_eq by force
  have disconnected_nodes_eq:
   "⋀document_ptr disconnected_nodes. document_ptr ≠ owner_document
      ⟹ h ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes
        = h' ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes"
    apply(rule reads_writes_preserved[OF get_disconnected_nodes_reads remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_child_nodes_get_disconnected_nodes set_disconnected_nodes_get_disconnected_nodes_different_pointers
    by (metis (no_types, lifting) Un_iff owner_document select_result_I2)
  then have disconnected_nodes_eq2:
    "⋀document_ptr. document_ptr ≠ owner_document
      ⟹ |h ⊢ get_disconnected_nodes document_ptr|_r = |h' ⊢ get_disconnected_nodes document_ptr|_r"
    using select_result_eq by force
```

```
  have "h2 ⊢ get_child_nodes ptr →r children_h"
    apply(rule reads_writes_separate_forwards[OF get_child_nodes_reads set_disconnected_nodes_writes h2
children_h] )
    by (simp add: set_disconnected_nodes_get_child_nodes)

  have "known_ptr ptr"
    using assms(3)
    using children_h get_child_nodes_ptr_in_heap local.known_ptrs_known_ptr by blast
  have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h2]
    using set_disconnected_nodes_types_preserved type_wf
    by(auto simp add: reflp_def transp_def)
  then have "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_child_nodes_writes h']
    using  set_child_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  have children_h': "h' ⊢ get_child_nodes ptr →r remove1 child children_h"
    using assms(2) owner_document h2 disconnected_nodes_h children_h
    apply(auto simp add: remove_child_def split: if_splits)[1]
    apply(drule bind_returns_heap_E3)
      apply(auto split: if_splits)[1]
     apply(simp)
    apply(auto split: if_splits)[1]
    apply(drule bind_returns_heap_E3)
      apply(auto)[1]
     apply(simp)
    apply(drule bind_returns_heap_E3)
      apply(auto)[1]
     apply(simp)
    apply(drule bind_returns_heap_E4)
      apply(auto)[1]
     apply(simp)
    using ‹type_wf h2› set_child_nodes_get_child_nodes ‹known_ptr ptr› h'
    by blast

  fix parent child
  assume a1: "(parent, child) ∈ parent_child_rel h'"
  then show "(parent, child) ∈ parent_child_rel h"
  proof (cases "parent = ptr")
    case True
    then show ?thesis
      using a1 remove_child_removes_parent[OF assms(1) assms(2)] children_h children_h'
            get_child_nodes_ptr_in_heap
      apply(auto simp add: parent_child_rel_def object_ptr_kinds_eq )[1]
      by (metis notin_set_remove1)
  next
    case False
    then show ?thesis
      using a1
      by(auto simp add: parent_child_rel_def object_ptr_kinds_eq3 children_eq2)
  qed
qed


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 -
```

```
  obtain owner_document children_h h2 disconnected_nodes_h where
    owner_document: "h ⊢ get_owner_document (cast_node_ptr2object_ptr child) →_r owner_document" and
    children_h: "h ⊢ get_child_nodes ptr →_r children_h" and
    child_in_children_h: "child ∈ set children_h" and
    disconnected_nodes_h: "h ⊢  get_disconnected_nodes owner_document →_r disconnected_nodes_h" and
    h2: "h ⊢  set_disconnected_nodes owner_document (child # disconnected_nodes_h) →_h h2" and
    h': "h2 ⊢ set_child_nodes ptr (remove1 child children_h) →_h h'"
    using assms(2)
    apply(auto simp add: remove_child_def elim!: bind_returns_heap_E
              dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                     pure_returns_heap_eq[rotated, OF get_child_nodes_pure] split: if_splits)[1]
    using pure_returns_heap_eq by fastforce

  have object_ptr_kinds_eq3: "object_ptr_kinds h = object_ptr_kinds h'"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_eq: "⋀ptrs. h ⊢ object_ptr_kinds_M →_r ptrs = h' ⊢ object_ptr_kinds_M →_r
ptrs"
    unfolding object_ptr_kinds_M_defs by simp
  then have object_ptr_kinds_eq2: "|h ⊢ object_ptr_kinds_M|_r = |h' ⊢ object_ptr_kinds_M|_r"
    using select_result_eq by force
  then have node_ptr_kinds_eq2: "|h ⊢ node_ptr_kinds_M|_r = |h' ⊢ node_ptr_kinds_M|_r"
    using node_ptr_kinds_M_eq by auto
  then have node_ptr_kinds_eq3: "node_ptr_kinds h = node_ptr_kinds h'"
    using node_ptr_kinds_M_eq by auto
  have document_ptr_kinds_eq2: "|h ⊢ document_ptr_kinds_M|_r = |h' ⊢ document_ptr_kinds_M|_r"
    using object_ptr_kinds_eq2 document_ptr_kinds_M_eq by auto
  then have document_ptr_kinds_eq3: "document_ptr_kinds h = document_ptr_kinds h'"
    using document_ptr_kinds_M_eq by auto
  have children_eq:
    "⋀ptr' children. ptr ≠ ptr' ⟹ h ⊢ get_child_nodes ptr' →_r children = h' ⊢ get_child_nodes ptr'
→_r children"
    apply(rule reads_writes_preserved[OF get_child_nodes_reads remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_disconnected_nodes_get_child_nodes set_child_nodes_get_child_nodes_different_pointers
    by fast
  then have children_eq2:
      "⋀ptr' children. ptr ≠ ptr' ⟹ |h ⊢ get_child_nodes ptr'|_r = |h' ⊢ get_child_nodes ptr'|_r"
    using select_result_eq by force
  have disconnected_nodes_eq: "⋀document_ptr disconnected_nodes. document_ptr ≠ owner_document
        ⟹ h ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes
           = h' ⊢ get_disconnected_nodes document_ptr →_r disconnected_nodes"
    apply(rule reads_writes_preserved[OF get_disconnected_nodes_reads remove_child_writes assms(2)])
    unfolding remove_child_locs_def
    using set_child_nodes_get_disconnected_nodes set_disconnected_nodes_get_disconnected_nodes_different_pointers
    by (metis (no_types, lifting) Un_iff owner_document select_result_I2)
  then have disconnected_nodes_eq2:
   "⋀document_ptr. document_ptr ≠ owner_document
    ⟹ |h ⊢ get_disconnected_nodes document_ptr|_r = |h' ⊢ get_disconnected_nodes document_ptr|_r"
    using select_result_eq by force

  have "h2 ⊢ get_child_nodes ptr →_r children_h"
    apply(rule reads_writes_separate_forwards[OF get_child_nodes_reads set_disconnected_nodes_writes h2
children_h] )
    by (simp add: set_disconnected_nodes_get_child_nodes)

  show "known_ptrs h'"
    using object_ptr_kinds_eq3 known_ptrs_preserved ‹known_ptrs h› by blast

  have "known_ptr ptr"
```

```
    using assms(3)
    using children_h get_child_nodes_ptr_in_heap local.known_ptrs_known_ptr by blast
have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h2]
    using set_disconnected_nodes_types_preserved type_wf
    by(auto simp add: reflp_def transp_def)
  then show "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_child_nodes_writes h']
    using  set_child_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  have children_h': "h' ⊢ get_child_nodes ptr →r remove1 child children_h"
    using assms(2) owner_document h2 disconnected_nodes_h children_h
    apply(auto simp add: remove_child_def split: if_splits)[1]
    apply(drule bind_returns_heap_E3)
      apply(auto split: if_splits)[1]
     apply(simp)
    apply(auto split: if_splits)[1]
    apply(drule bind_returns_heap_E3)
      apply(auto)[1]
     apply(simp)
    apply(drule bind_returns_heap_E3)
      apply(auto)[1]
     apply(simp)
    apply(drule bind_returns_heap_E4)
      apply(auto)[1]
     apply simp
    using ‹type_wf h2› set_child_nodes_get_child_nodes ‹known_ptr ptr› h'
    by blast

  have disconnected_nodes_h2: "h2 ⊢ get_disconnected_nodes owner_document →r child # disconnected_nodes_h"
    using owner_document assms(2) h2 disconnected_nodes_h
    apply (auto simp add: remove_child_def  split: if_splits)[1]
    apply(drule bind_returns_heap_E2)
      apply(auto split: if_splits)[1]
     apply(simp)
    by(auto simp add: local.set_disconnected_nodes_get_disconnected_nodes split: if_splits)
  then have disconnected_nodes_h': "h' ⊢ get_disconnected_nodes owner_document →r child # disconnected_nodes_h"
    apply(rule reads_writes_separate_forwards[OF get_disconnected_nodes_reads set_child_nodes_writes h'])
    by (simp add: set_child_nodes_get_disconnected_nodes)

  moreover have "a_acyclic_heap h"
    using assms(1)  by (simp add: heap_is_wellformed_def)
  have "parent_child_rel h' ⊆ parent_child_rel h"
  proof (standard, safe)
    fix parent child
    assume a1: "(parent, child) ∈ parent_child_rel h'"
    then show "(parent, child) ∈ parent_child_rel h"
    proof (cases "parent = ptr")
      case True
      then show ?thesis
        using a1 remove_child_removes_parent[OF assms(1) assms(2)] children_h children_h'
              get_child_nodes_ptr_in_heap
        apply(auto simp add: parent_child_rel_def object_ptr_kinds_eq )[1]
        by (metis imageI notin_set_remove1)
    next
      case False
      then show ?thesis
        using a1
        by(auto simp add: parent_child_rel_def object_ptr_kinds_eq3 children_eq2)
    qed
  qed
```

```
then have "a_acyclic_heap h'"
  using ‹a_acyclic_heap h› acyclic_heap_def acyclic_subset by blast

moreover have "a_all_ptrs_in_heap h"
  using assms(1)  by (simp add: heap_is_wellformed_def)
then have "a_all_ptrs_in_heap h'"
  apply(auto simp add: a_all_ptrs_in_heap_def node_ptr_kinds_eq3 disconnected_nodes_eq)[1]
   apply (metis (no_types, lifting) type_wf assms(3) children_eq2 children_h children_h'
                fset_of_list_subset fsubsetD get_child_nodes_ok get_child_nodes_ptr_in_heap
                is_OK_returns_result_E is_OK_returns_result_I local.known_ptrs_known_ptr
                object_ptr_kinds_eq3 select_result_I2 set_remove1_subset)
  by (metis (no_types, lifting)
     ‹⋀thesis. (⋀owner_document children_h h2 disconnected_nodes_h.
           ⟦h ⊢ get_owner_document (cast_node_ptr2object_ptr child) →_r owner_document;
            h ⊢ get_child_nodes ptr →_r children_h; child ∈ set children_h;
            h ⊢ get_disconnected_nodes owner_document →_r disconnected_nodes_h;
            h ⊢ set_disconnected_nodes owner_document (child # disconnected_nodes_h) →_h h2;
           h2 ⊢ set_child_nodes ptr (remove1 child children_h) →_h h'⟧ ⟹ thesis) ⟹ thesis›
     disconnected_nodes_h disconnected_nodes_eq disconnected_nodes_h' fset_mp fset_of_list_elem
     returns_result_eq set_ConsD)

moreover have "a_owner_document_valid h"
  using assms(1)  by (simp add: heap_is_wellformed_def)
then have "a_owner_document_valid h'"
  apply(auto simp add: a_owner_document_valid_def object_ptr_kinds_eq3 document_ptr_kinds_eq3
                       node_ptr_kinds_eq3)[1]
proof -
  fix node_ptr
  assume 0: "∀node_ptr. node_ptr |∈| 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)"
    and 1: "node_ptr |∈| node_ptr_kinds h'"
    and 2: "∀parent_ptr. parent_ptr |∈| object_ptr_kinds h'
                  ⟶ node_ptr ∉ set |h' ⊢ get_child_nodes parent_ptr|_r"
  then show "∃document_ptr. document_ptr |∈| document_ptr_kinds h'
                    ∧ node_ptr ∈ set |h' ⊢ get_disconnected_nodes document_ptr|_r"
  proof (cases "node_ptr = child")
    case True
    show ?thesis
      apply(rule exI[where x=owner_document])
      using children_eq2 disconnected_nodes_eq2 children_h children_h' disconnected_nodes_h' True
      by (metis (no_types, lifting) get_disconnected_nodes_ptr_in_heap is_OK_returns_result_I
                                    list.set_intros(1) select_result_I2)
  next
    case False
    then show ?thesis
      using 0 1 2 children_eq2 children_h children_h' disconnected_nodes_eq2 disconnected_nodes_h
            disconnected_nodes_h'
      apply(auto simp add: children_eq2 disconnected_nodes_eq2 dest!: select_result_I2)[1]
      by (metis children_eq2 disconnected_nodes_eq2 in_set_remove1 list.set_intros(2))
  qed
qed

moreover
{
  have h0: "a_distinct_lists h"
    using assms(1)  by (simp add: heap_is_wellformed_def)
  moreover have ha1: "(⋃x∈set |h ⊢ object_ptr_kinds_M|_r. set |h ⊢ get_child_nodes x|_r)
              ∩ (⋃x∈set |h ⊢ document_ptr_kinds_M|_r. set |h ⊢ get_disconnected_nodes x|_r) = {}"
    using ‹a_distinct_lists h›
    unfolding a_distinct_lists_def
```

```
    by(auto)
  have ha2: "ptr |∈| object_ptr_kinds h"
    using children_h get_child_nodes_ptr_in_heap by blast
  have ha3: "child ∈ set |h ⊢ get_child_nodes ptr|_r"
    using child_in_children_h children_h
    by(simp)
  have child_not_in: "⋀document_ptr. document_ptr |∈| document_ptr_kinds h
                      ⟹ child ∉ set |h ⊢ get_disconnected_nodes document_ptr|_r"
    using ha1 ha2 ha3
    apply(simp)
    using IntI by fastforce
  moreover have "distinct |h ⊢ object_ptr_kinds_M|_r"
    apply(rule select_result_I)
    by(auto simp add: object_ptr_kinds_M_defs)
  moreover have "distinct |h ⊢ document_ptr_kinds_M|_r"
    apply(rule select_result_I)
    by(auto simp add: document_ptr_kinds_M_defs)
  ultimately have "a_distinct_lists h'"
  proof(simp (no_asm) add: a_distinct_lists_def, safe)
    assume 1: "a_distinct_lists h"
      and 3: "distinct |h ⊢ object_ptr_kinds_M|_r"

    assume 1: "a_distinct_lists h"
      and 3: "distinct |h ⊢ object_ptr_kinds_M|_r"
    have 4: "distinct (concat ((map (λptr. |h ⊢ get_child_nodes ptr|_r) |h ⊢ object_ptr_kinds_M|_r)))"
      using 1 by(auto simp add: a_distinct_lists_def)
    show "distinct (concat (map (λptr. |h' ⊢ get_child_nodes ptr|_r)
                  (sorted_list_of_set (fset (object_ptr_kinds h')))))"
    proof(rule distinct_concat_map_I[OF 3[unfolded object_ptr_kinds_eq2], simplified])
      fix x
      assume 5: "x |∈| object_ptr_kinds h'"
      then have 6: "distinct |h ⊢ get_child_nodes x|_r"
        using 4 distinct_concat_map_E object_ptr_kinds_eq2 by fastforce
      obtain children where children: "h ⊢ get_child_nodes x →_r children"
                        and distinct_children: "distinct children"
        by (metis "5" "6" type_wf assms(3) get_child_nodes_ok local.known_ptrs_known_ptr
                          object_ptr_kinds_eq3 select_result_I)
      obtain children' where children': "h' ⊢ get_child_nodes x →_r children'"
        using children children_eq children_h' by fastforce
      then have "distinct children'"
      proof (cases "ptr = x")
        case True
        then show ?thesis
          using children distinct_children children_h children_h'
          by (metis children' distinct_remove1 returns_result_eq)
      next
        case False
        then show ?thesis
          using children distinct_children children_eq[OF False]
          using children' distinct_lists_children h0
          using select_result_I2 by fastforce
      qed

      then show "distinct |h' ⊢ get_child_nodes x|_r"
        using children' by(auto simp add: )
    next
      fix x y
      assume 5: "x |∈| object_ptr_kinds h'" and 6: "y |∈| object_ptr_kinds h'" and 7: "x ≠ y"
      obtain children_x where children_x: "h ⊢ get_child_nodes x →_r children_x"
        by (metis "5" type_wf assms(3) get_child_nodes_ok is_OK_returns_result_E
                  local.known_ptrs_known_ptr object_ptr_kinds_eq3)
      obtain children_y where children_y: "h ⊢ get_child_nodes y →_r children_y"
        by (metis "6" type_wf assms(3) get_child_nodes_ok is_OK_returns_result_E
```

```
            local.known_ptrs_known_ptr object_ptr_kinds_eq3)
    obtain children_x' where children_x': "h' ⊢ get_child_nodes x →r children_x'"
      using children_eq children_h' children_x by fastforce
    obtain children_y' where children_y': "h' ⊢ get_child_nodes y →r children_y'"
      using children_eq children_h' children_y by fastforce
    have "distinct (concat (map (λptr. |h ⊢ get_child_nodes ptr|r) |h ⊢ object_ptr_kinds_M|r))"
      using h0 by(auto simp add: a_distinct_lists_def)
    then have 8: "set children_x ∩ set children_y = {}"
      using "7" assms(1) children_x children_y local.heap_is_wellformed_one_parent by blast
    have "set children_x' ∩ set children_y' = {}"
    proof (cases "ptr = x")
      case True
      then have "ptr ≠ y"
        by(simp add: 7)
      have "children_x' = remove1 child children_x"
        using children_h children_h' children_x children_x' True returns_result_eq by fastforce
      moreover have "children_y' = children_y"
        using children_y children_y' children_eq[OF ⟨ptr ≠ y⟩] by auto
      ultimately show ?thesis
        using 8 set_remove1_subset by fastforce
    next
      case False
      then show ?thesis
      proof (cases "ptr = y")
        case True
        have "children_y' = remove1 child children_y"
          using children_h children_h' children_y children_y' True returns_result_eq by fastforce
        moreover have "children_x' = children_x"
          using children_x children_x' children_eq[OF ⟨ptr ≠ x⟩] by auto
        ultimately show ?thesis
          using 8 set_remove1_subset by fastforce
      next
        case False
        have "children_x' = children_x"
          using children_x children_x' children_eq[OF ⟨ptr ≠ x⟩] by auto
        moreover have "children_y' = children_y"
          using children_y children_y' children_eq[OF ⟨ptr ≠ y⟩] by auto
        ultimately show ?thesis
          using 8 by simp
      qed
    qed
    then show "set |h' ⊢ get_child_nodes x|r ∩ set |h' ⊢ get_child_nodes y|r = {}"
      using children_x' children_y'
      by (metis (no_types, lifting) select_result_I2)
  qed
next
  assume 2: "distinct |h ⊢ document_ptr_kinds_M|r"
  then have 4: "distinct  (sorted_list_of_set (fset (document_ptr_kinds h')))"
    by simp
  have 3: "distinct (concat (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|r)
                  (sorted_list_of_set (fset (document_ptr_kinds h')))))"
    using h0
    by(simp add: a_distinct_lists_def document_ptr_kinds_eq3)

  show "distinct (concat (map (λdocument_ptr. |h' ⊢ get_disconnected_nodes document_ptr|r)
                    (sorted_list_of_set (fset (document_ptr_kinds h')))))"
  proof(rule distinct_concat_map_I[OF 4[unfolded document_ptr_kinds_eq3]])
    fix x
    assume 4: "x ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))"
    have 5: "distinct |h ⊢ get_disconnected_nodes x|r"
      using distinct_lists_disconnected_nodes[OF h0] 4 get_disconnected_nodes_ok
      by (simp add: type_wf document_ptr_kinds_eq3 select_result_I)
    show "distinct |h' ⊢ get_disconnected_nodes x|r"
```

```
    proof (cases "x = owner_document")
      case True
      have "child ∉ set |h ⊢ get_disconnected_nodes x|_r"
        using child_not_in  document_ptr_kinds_eq2 "4" by fastforce
      moreover have "|h' ⊢ get_disconnected_nodes x|_r = child # |h ⊢ get_disconnected_nodes x|_r"
        using disconnected_nodes_h' disconnected_nodes_h unfolding True
        by(simp)
      ultimately show ?thesis
        using 5 unfolding True
        by simp
    next
      case False
      show ?thesis
        using "5" False disconnected_nodes_eq2 by auto
    qed
  next
    fix x y
    assume 4: "x ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))"
      and 5: "y ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))" and "x ≠ y"
    obtain disc_nodes_x where disc_nodes_x: "h ⊢ get_disconnected_nodes x →_r disc_nodes_x"
      using 4 get_disconnected_nodes_ok[OF ‹type_wf h›, of x] document_ptr_kinds_eq2
      by auto
    obtain disc_nodes_y where disc_nodes_y: "h ⊢ get_disconnected_nodes y →_r disc_nodes_y"
      using 5 get_disconnected_nodes_ok[OF ‹type_wf h›, of y] document_ptr_kinds_eq2
      by auto
    obtain disc_nodes_x' where disc_nodes_x': "h' ⊢ get_disconnected_nodes x →_r disc_nodes_x'"
      using 4 get_disconnected_nodes_ok[OF ‹type_wf h'›, of x] document_ptr_kinds_eq2
      by auto
    obtain disc_nodes_y' where disc_nodes_y': "h' ⊢ get_disconnected_nodes y →_r disc_nodes_y'"
      using 5 get_disconnected_nodes_ok[OF ‹type_wf h'›, of y] document_ptr_kinds_eq2
      by auto
    have "distinct
          (concat (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|_r) |h ⊢ document_ptr_kinds_M
      using h0 by (simp add: a_distinct_lists_def)
    then have 6: "set disc_nodes_x ∩ set disc_nodes_y = {}"
      using ‹x ≠ y› assms(1) disc_nodes_x disc_nodes_y local.heap_is_wellformed_one_disc_parent
      by blast

    have "set disc_nodes_x' ∩ set disc_nodes_y' = {}"
    proof (cases "x = owner_document")
      case True
      then have "y ≠ owner_document"
        using ‹x ≠ y› by simp
      then have "disc_nodes_y' = disc_nodes_y"
        using disconnected_nodes_eq[OF ‹y ≠ owner_document›] disc_nodes_y disc_nodes_y'
        by auto
      have "disc_nodes_x' = child # disc_nodes_x"
        using disconnected_nodes_h' disc_nodes_x disc_nodes_x' True disconnected_nodes_h returns_result_eq

        by fastforce
      have "child ∉ set disc_nodes_y"
        using child_not_in  disc_nodes_y 5
        using document_ptr_kinds_eq2  by fastforce
      then show ?thesis
        apply(unfold ‹disc_nodes_x' = child # disc_nodes_x› ‹disc_nodes_y' = disc_nodes_y›)
        using 6 by auto
    next
      case False
      then show ?thesis
      proof (cases "y = owner_document")
        case True
        then have "disc_nodes_x' = disc_nodes_x"
          using disconnected_nodes_eq[OF ‹x ≠ owner_document›] disc_nodes_x disc_nodes_x' by auto
```

```
        have "disc_nodes_y' = child # disc_nodes_y"
          using disconnected_nodes_h' disc_nodes_y disc_nodes_y' True disconnected_nodes_h returns_result_eq

          by fastforce
        have "child ∉ set disc_nodes_x"
          using child_not_in  disc_nodes_x 4
          using document_ptr_kinds_eq2  by fastforce
        then show ?thesis
          apply(unfold ⟨disc_nodes_y' = child # disc_nodes_y⟩ ⟨disc_nodes_x' = disc_nodes_x⟩)
          using 6 by auto
      next
        case False
        have "disc_nodes_x' = disc_nodes_x"
          using disconnected_nodes_eq[OF ⟨x ≠ owner_document⟩] disc_nodes_x disc_nodes_x' by auto
        have "disc_nodes_y' = disc_nodes_y"
          using disconnected_nodes_eq[OF ⟨y ≠ owner_document⟩] disc_nodes_y disc_nodes_y' by auto
        then show ?thesis
          apply(unfold ⟨disc_nodes_y' = disc_nodes_y⟩ ⟨disc_nodes_x' = disc_nodes_x⟩)
          using 6 by auto
      qed
    qed
    then show "set |h' ⊢ get_disconnected_nodes x|_r ∩ set |h' ⊢ get_disconnected_nodes y|_r = {}"
      using disc_nodes_x' disc_nodes_y' by auto
  qed
next
fix x xa xb
assume 1: "xa ∈ fset (object_ptr_kinds h')"
  and 2: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
  and 3: "xb ∈ fset (document_ptr_kinds h')"
  and 4: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
  obtain disc_nodes where disc_nodes: "h ⊢ get_disconnected_nodes xb →_r disc_nodes"
    using 3 get_disconnected_nodes_ok[OF ⟨type_wf h⟩, of xb] document_ptr_kinds_eq2 by auto
  obtain disc_nodes' where disc_nodes': "h' ⊢ get_disconnected_nodes xb →_r disc_nodes'"
    using 3 get_disconnected_nodes_ok[OF ⟨type_wf h'⟩, of xb]  document_ptr_kinds_eq2 by auto

  obtain children where children: "h ⊢ get_child_nodes xa →_r children"
    by (metis "1" type_wf assms(3) finite_set_in get_child_nodes_ok is_OK_returns_result_E
              local.known_ptrs_known_ptr object_ptr_kinds_eq3)
  obtain children' where children': "h' ⊢ get_child_nodes xa →_r children'"
    using children children_eq children_h' by fastforce
  have "⋀x. x ∈ set |h ⊢ get_child_nodes xa|_r ⟹ x ∈ set |h ⊢ get_disconnected_nodes xb|_r ⟹ False"
    using 1 3
    apply(fold ⟨ object_ptr_kinds h =  object_ptr_kinds h'⟩)
    apply(fold ⟨ document_ptr_kinds h =  document_ptr_kinds h'⟩)
    using children disc_nodes h0 apply(auto simp add: a_distinct_lists_def)[1]
  by (metis (no_types, lifting) h0 local.distinct_lists_no_parent select_result_I2)
  then have 5: "⋀x. x ∈ set children ⟹ x ∈ set disc_nodes ⟹ False"
    using children disc_nodes  by fastforce
  have 6: "|h' ⊢ get_child_nodes xa|_r = children'"
    using children' by (simp add: )
  have 7: "|h' ⊢ get_disconnected_nodes xb|_r = disc_nodes'"
    using disc_nodes' by (simp add: )
  have "False"
  proof (cases "xa = ptr")
    case True
    have "distinct children_h"
      using children_h distinct_lists_children h0 ⟨known_ptr ptr⟩ by blast
    have "|h' ⊢ get_child_nodes ptr|_r = remove1 child children_h"
      using children_h'
      by(simp add: )
    have "children = children_h"
      using True children children_h by auto
    show ?thesis
```

```
      using disc_nodes' children' 5 2 4 children_h ‹distinct children_h› disconnected_nodes_h'
      apply(auto simp add: 6 7
            ‹xa = ptr› ‹|h' ⊢ get_child_nodes ptr|_r = remove1 child children_h› ‹children = children_h›)[1]
      by (metis (no_types, lifting) disc_nodes disconnected_nodes_eq2 disconnected_nodes_h
                                    select_result_I2 set_ConsD)
    next
      case False
      have "children' = children"
        using children' children children_eq[OF False[symmetric]]
        by auto
      then show ?thesis
      proof (cases "xb = owner_document")
        case True
        then show ?thesis
          using disc_nodes disconnected_nodes_h disconnected_nodes_h'
          using "2" "4" "5" "6" "7" False ‹children' = children› assms(1) child_in_children_h
                child_parent_dual children children_h disc_nodes' get_child_nodes_ptr_in_heap
                list.set_cases list.simps(3) option.simps(1) returns_result_eq set_ConsD
          by (metis (no_types, hide_lams) assms(3) type_wf)
      next
        case False
        then show ?thesis
          using "2" "4" "5" "6" "7" ‹children' = children› disc_nodes disc_nodes'
                disconnected_nodes_eq returns_result_eq
          by metis
      qed
    qed
    then show "x ∈ {}"
      by simp
  qed
}

ultimately show "heap_is_wellformed h'"
  using heap_is_wellformed_def by blast
qed

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 -
  obtain owner_document children_h h2 disconnected_nodes_h where
    owner_document: "h ⊢ get_owner_document (cast_{node_ptr2object_ptr} child) →_r owner_document" and
    children_h: "h ⊢ get_child_nodes ptr' →_r children_h" and
    child_in_children_h: "child ∈ set children_h" and
    disconnected_nodes_h: "h ⊢  get_disconnected_nodes owner_document →_r disconnected_nodes_h" and
    h2: "h ⊢  set_disconnected_nodes owner_document (child # disconnected_nodes_h) →_h h2" and
    h': "h2 ⊢ set_child_nodes ptr' (remove1 child children_h) →_h h'"
    using assms(2)
    apply(auto simp add: remove_child_def elim!: bind_returns_heap_E
              dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                     pure_returns_heap_eq[rotated, OF get_child_nodes_pure]
              split: if_splits)[1]
    using pure_returns_heap_eq
    by fastforce
  have "object_ptr_kinds h = object_ptr_kinds h'"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF remove_child_writes remove_child])
    unfolding remove_child_locs_def
    using set_child_nodes_pointers_preserved set_disconnected_nodes_pointers_preserved
```

```
    by (auto simp add: reflp_def transp_def)
  moreover have "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF remove_child_writes assms(2)]
    using set_child_nodes_types_preserved set_disconnected_nodes_types_preserved type_wf
    unfolding remove_child_locs_def
    apply(auto simp add: reflp_def transp_def)
    by blast
  ultimately show ?thesis
    using remove_child_removes_parent remove_child_heap_is_wellformed_preserved child_parent_dual
    by (meson children known_ptrs local.known_ptrs_preserved option.distinct(1) remove_child
          returns_result_eq type_wf wellformed)
qed

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 -
  obtain h2 disc_nodes owner_document where
    "h ⊢ get_owner_document (cast node_ptr) →r owner_document" and
    "h ⊢ get_disconnected_nodes owner_document →r disc_nodes" and
    h2: "h ⊢ set_disconnected_nodes owner_document (node_ptr # disc_nodes) →h h2" and
    "h2 ⊢ set_child_nodes ptr children →h h'"
    using assms(5)
    apply(auto simp add: remove_child_def
           dest!: bind_returns_heap_E3[rotated, OF assms(4) get_child_nodes_pure, rotated])[1]
    by(auto elim!: bind_returns_heap_E
                   bind_returns_heap_E2[rotated,OF get_owner_document_pure, rotated]
                   bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated])
  have "known_ptr ptr"
    by (meson assms(3) assms(4) is_OK_returns_result_I get_child_nodes_ptr_in_heap known_ptrs_known_ptr)
  moreover have "h2 ⊢ get_child_nodes ptr →r node_ptr # children"
    apply(rule reads_writes_separate_forwards[OF get_child_nodes_reads set_disconnected_nodes_writes h2
assms(4)])
    using set_disconnected_nodes_get_child_nodes set_child_nodes_get_child_nodes_different_pointers
    by fast
  moreover have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h2]
    using ‹type_wf h› set_disconnected_nodes_types_preserved
    by(auto simp add:  reflp_def transp_def)
  ultimately show ?thesis
    using set_child_nodes_get_child_nodes‹h2 ⊢ set_child_nodes ptr children →h h'›
    by fast
qed

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 -
  have "h ⊢ get_parent node_ptr →r Some ptr"
    using child_parent_dual assms by fastforce
  show ?thesis
    using assms remove_child_removes_first_child
    by(auto simp add: remove_def
           dest!: bind_returns_heap_E3[rotated, OF ‹h ⊢ get_parent node_ptr →r Some ptr›, rotated]
                  bind_returns_heap_E3[rotated, OF assms(4) get_child_nodes_pure, rotated])
qed

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 []"
  using assms
proof(induct children arbitrary: h h')
  case Nil
  then show ?case
    by simp
next
  case (Cons a children)
  have "h ⊢ get_parent a →r Some ptr"
    using child_parent_dual Cons by fastforce
  with Cons show ?case
  proof(auto elim!: bind_returns_heap_E)[1]
    fix h2
    assume 0: "(⋀h h'. 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 [])"
      and 1: "heap_is_wellformed h"
      and 2: "type_wf h"
      and 3: "known_ptrs h"
      and 4: "h ⊢ get_child_nodes ptr →r a # children"
      and 5: "h ⊢ get_parent a →r Some ptr"
      and 7: "h ⊢ remove a →h h2"
      and 8: "h2 ⊢ forall_M remove children →h h'"
    then have "h2 ⊢ get_child_nodes ptr →r children"
      using remove_removes_child by blast

    moreover have "heap_is_wellformed h2"
      using 7 1 2 3 remove_child_heap_is_wellformed_preserved(3)
      by(auto simp add: remove_def
               elim!: bind_returns_heap_E
                      bind_returns_heap_E2[rotated, OF get_parent_pure, rotated]
               split: option.splits)
    moreover have "type_wf h2"
      using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF remove_writes 7]
      using ⟨type_wf h⟩ remove_child_types_preserved
      by(auto simp add: a_remove_child_locs_def reflp_def transp_def)
    moreover have "object_ptr_kinds h = object_ptr_kinds h2"
      using 7
      apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                  OF remove_writes])
      using remove_child_pointers_preserved
      by (auto simp add: reflp_def transp_def)
    then have "known_ptrs h2"
      using 3  known_ptrs_preserved by blast

    ultimately show "h' ⊢ get_child_nodes ptr →r []"
      using 0 8 by fast
  qed
qed
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_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_wf2_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
    heap_is_wellformed parent_child_rel
  by unfold_locales

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"
  apply(auto simp add: l_remove_child_wf2_def l_remove_child_wf2_axioms_def instances)[1]
  using remove_child_heap_is_wellformed_preserved apply(fast, fast, fast)
  using remove_child_removes_child apply fast
  using remove_child_removes_first_child apply fast
  using remove_removes_child apply fast
  using remove_for_all_empty_children apply fast
  done
```

6.3.10 adopt_node

```
locale l_adopt_node_wf_Core_DOM =
  l_adopt_node_Core_DOM +
  l_get_parent_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 -
  obtain old_document parent_opt h2 where
    old_document: "h ⊢ get_owner_document (cast node) →_r old_document" and
    parent_opt: "h ⊢ get_parent node →_r parent_opt" and
    h2: "h ⊢ (case parent_opt of Some parent ⇒ do { remove_child parent node }
                               | None ⇒ do { return ()}) →_h h2" and
    h': "h2 ⊢ (if owner_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 owner_document;
        set_disconnected_nodes owner_document (node # disc_nodes)
      } else do { return () }) →_h h'"
    using assms(4)
    by(auto simp add: adopt_node_def elim!: bind_returns_heap_E
```

```
            dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                   pure_returns_heap_eq[rotated, OF get_parent_pure])
  have "h2 ⊢ get_child_nodes ptr' →r children"
    using h2 remove_child_removes_first_child assms(1) assms(2) assms(3) assms(5)
    by (metis list.set_intros(1) local.child_parent_dual option.simps(5) parent_opt returns_result_eq)
  then
  show ?thesis
    using h'
    by(auto elim!: bind_returns_heap_E bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated]

            dest!: reads_writes_separate_forwards[OF get_child_nodes_reads set_disconnected_nodes_writes]

            split: if_splits)
qed
end

locale l_adopt_node_wf2Core_DOM =
  l_adopt_node_wfCore_DOM +
  l_adopt_nodeCore_DOM +
  l_get_parent_wfCore_DOM +
  l_get_root_node +
  l_get_owner_document_wf +
  l_remove_child_wf2 +
  l_heap_is_wellformedCore_DOM
begin

lemma adopt_node_removes_child:
  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 -
  obtain old_document parent_opt h' where
    old_document: "h ⊢ get_owner_document (cast node_ptr) →r old_document" and
    parent_opt: "h ⊢ get_parent node_ptr →r parent_opt" and
    h': "h ⊢ (case parent_opt of Some parent ⇒ remove_child parent node_ptr | None ⇒ return () ) →h
h'"
    using adopt_node get_parent_pure
    by(auto simp add: adopt_node_def
            elim!: bind_returns_heap_E bind_returns_heap_E2[rotated, OF get_owner_document_pure, rotated]

                   bind_returns_heap_E2[rotated, OF get_parent_pure, rotated]
                   bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated]
            split: if_splits)

  then have "h' ⊢ get_child_nodes ptr →r children"
    using adopt_node
    apply(auto simp add: adopt_node_def
                  dest!: bind_returns_heap_E3[rotated, OF old_document, rotated]
                         bind_returns_heap_E3[rotated, OF parent_opt, rotated]
                  elim!: bind_returns_heap_E4[rotated, OF h', rotated])[1]
    apply(auto split: if_splits
              elim!: bind_returns_heap_E
                     bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated])[1]
     apply (simp add: set_disconnected_nodes_get_child_nodes children
                      reads_writes_preserved[OF get_child_nodes_reads set_disconnected_nodes_writes])
    using children by blast
  show ?thesis
  proof(insert parent_opt h', induct parent_opt)
    case None
    then show ?case
```

```
    using child_parent_dual wellformed known_ptrs type_wf
          ⟨h' ⊢ get_child_nodes ptr →r children⟩ returns_result_eq
    by fastforce
  next
    case (Some option)
    then show ?case
    using remove_child_removes_child ⟨h' ⊢ get_child_nodes ptr →r children⟩ known_ptrs type_wf
          wellformed
    by auto
  qed
qed

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'"
proof -
  obtain old_document parent_opt h2 where
    old_document: "h ⊢ get_owner_document (cast child) →r old_document"
  and
    parent_opt: "h ⊢ get_parent child →r parent_opt"
  and
    h2: "h ⊢ (case parent_opt of Some parent ⇒ remove_child parent child | None ⇒ return ()) →h h2"

  and
    h': "h2 ⊢ (if document_ptr ≠ old_document then do {
        old_disc_nodes ← get_disconnected_nodes old_document;
        set_disconnected_nodes old_document (remove1 child old_disc_nodes);
        disc_nodes ← get_disconnected_nodes document_ptr;
        set_disconnected_nodes document_ptr (child # disc_nodes)
      } else do {
        return ()
      }) →h h'"
    using assms(2)
    by(auto simp add: adopt_node_def elim!: bind_returns_heap_E
            dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                   pure_returns_heap_eq[rotated, OF get_parent_pure])

  have object_ptr_kinds_h_eq3: "object_ptr_kinds h = object_ptr_kinds h2"
    using h2 apply(simp split: option.splits)
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF remove_child_writes])
    using remove_child_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_M_eq_h:
          "⋀ptrs. h ⊢ object_ptr_kinds_M →r ptrs = h2 ⊢ object_ptr_kinds_M →r ptrs"
    unfolding object_ptr_kinds_M_defs by simp
  then have object_ptr_kinds_eq_h: "|h ⊢ object_ptr_kinds_M|r = |h2 ⊢ object_ptr_kinds_M|r"
    by simp
  then have node_ptr_kinds_eq_h: "|h ⊢ node_ptr_kinds_M|r = |h2 ⊢ node_ptr_kinds_M|r"
    using node_ptr_kinds_M_eq by blast

  have wellformed_h2: "heap_is_wellformed h2"
    using h2 remove_child_heap_is_wellformed_preserved known_ptrs type_wf
    by (metis (no_types, lifting) assms(1) option.case_eq_if pure_returns_heap_eq return_pure)
  then show ?thesis
  proof(cases "document_ptr = old_document")
    case True
    then show ?thesis
      using h' wellformed_h2 by auto
  next
```

```
case False
then obtain h3 old_disc_nodes disc_nodes_document_ptr_h3 where
  docs_neq: "document_ptr ≠ old_document" and
  old_disc_nodes: "h2 ⊢ get_disconnected_nodes old_document →r old_disc_nodes" and
  h3: "h2 ⊢ set_disconnected_nodes old_document (remove1 child old_disc_nodes) →h h3" and
  disc_nodes_document_ptr_h3:
        "h3 ⊢ get_disconnected_nodes document_ptr →r disc_nodes_document_ptr_h3" and
  h': "h3 ⊢ set_disconnected_nodes document_ptr (child # disc_nodes_document_ptr_h3) →h h'"
  using h'
  by(auto elim!: bind_returns_heap_E
                 bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated] )

have object_ptr_kinds_h2_eq3: "object_ptr_kinds h2 = object_ptr_kinds h3"
  apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                              OF set_disconnected_nodes_writes h3])
  using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
  by (auto simp add: reflp_def transp_def)
then have object_ptr_kinds_M_eq_h2:
         "⋀ptrs. h2 ⊢ object_ptr_kinds_M →r ptrs = h3 ⊢ object_ptr_kinds_M →r ptrs"
  by(simp add: object_ptr_kinds_M_defs)
then have object_ptr_kinds_eq_h2: "|h2 ⊢ object_ptr_kinds_M|r = |h3 ⊢ object_ptr_kinds_M|r"
  by(simp)
then have node_ptr_kinds_eq_h2: "|h2 ⊢ node_ptr_kinds_M|r = |h3 ⊢ node_ptr_kinds_M|r"
  using node_ptr_kinds_M_eq by blast
then have node_ptr_kinds_eq3_h2: "node_ptr_kinds h2 = node_ptr_kinds h3"
  by auto
have document_ptr_kinds_eq2_h2: "|h2 ⊢ document_ptr_kinds_M|r = |h3 ⊢ document_ptr_kinds_M|r"
  using object_ptr_kinds_eq_h2 document_ptr_kinds_M_eq by auto
then have document_ptr_kinds_eq3_h2: "document_ptr_kinds h2 = document_ptr_kinds h3"
  using object_ptr_kinds_eq_h2 document_ptr_kinds_M_eq by auto
have children_eq_h2:
  "⋀ptr children. h2 ⊢ get_child_nodes ptr →r children = h3 ⊢ get_child_nodes ptr →r children"
  using get_child_nodes_reads set_disconnected_nodes_writes h3
  apply(rule reads_writes_preserved)
  by (simp add: set_disconnected_nodes_get_child_nodes)
then have children_eq2_h2: "⋀ptr. |h2 ⊢ get_child_nodes ptr|r = |h3 ⊢ get_child_nodes ptr|r"
  using select_result_eq by force

have object_ptr_kinds_h3_eq3: "object_ptr_kinds h3 = object_ptr_kinds h'"
  apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                              OF set_disconnected_nodes_writes h'])
  using set_disconnected_nodes_pointers_preserved set_child_nodes_pointers_preserved
  by (auto simp add: reflp_def transp_def)
then have object_ptr_kinds_M_eq_h3:
  "⋀ptrs. h3 ⊢ object_ptr_kinds_M →r ptrs = h' ⊢ object_ptr_kinds_M →r ptrs"
  by(simp add: object_ptr_kinds_M_defs)
then have object_ptr_kinds_eq_h3: "|h3 ⊢ object_ptr_kinds_M|r = |h' ⊢ object_ptr_kinds_M|r"
  by(simp)
then have node_ptr_kinds_eq_h3: "|h3 ⊢ node_ptr_kinds_M|r = |h' ⊢ node_ptr_kinds_M|r"
  using node_ptr_kinds_M_eq by blast
then have node_ptr_kinds_eq3_h3: "node_ptr_kinds h3 = node_ptr_kinds h'"
  by auto
have document_ptr_kinds_eq2_h3: "|h3 ⊢ document_ptr_kinds_M|r = |h' ⊢ document_ptr_kinds_M|r"
  using object_ptr_kinds_eq_h3 document_ptr_kinds_M_eq by auto
then have document_ptr_kinds_eq3_h3: "document_ptr_kinds h3 = document_ptr_kinds h'"
  using object_ptr_kinds_eq_h3 document_ptr_kinds_M_eq by auto
have children_eq_h3:
  "⋀ptr children. h3 ⊢ get_child_nodes ptr →r children = h' ⊢ get_child_nodes ptr →r children"
  using get_child_nodes_reads set_disconnected_nodes_writes h'
  apply(rule reads_writes_preserved)
  by (simp add: set_disconnected_nodes_get_child_nodes)
then have children_eq2_h3: "⋀ptr. |h3 ⊢ get_child_nodes ptr|r = |h' ⊢ get_child_nodes ptr|r"
  using select_result_eq by force
```

```
    have disconnected_nodes_eq_h2:
      "⋀doc_ptr disc_nodes. old_document ≠ doc_ptr
      ⟹ h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes = h3 ⊢ get_disconnected_nodes doc_ptr →r
disc_nodes"
      using get_disconnected_nodes_reads set_disconnected_nodes_writes h3
      apply(rule reads_writes_preserved)
      by (simp add: set_disconnected_nodes_get_disconnected_nodes_different_pointers)
    then have disconnected_nodes_eq2_h2:
      "⋀doc_ptr. old_document ≠ doc_ptr
       ⟹ |h2 ⊢ get_disconnected_nodes doc_ptr|r = |h3 ⊢ get_disconnected_nodes doc_ptr|r"
      using select_result_eq by force
    obtain disc_nodes_old_document_h2 where disc_nodes_old_document_h2:
      "h2 ⊢ get_disconnected_nodes old_document →r disc_nodes_old_document_h2"
      using old_disc_nodes by blast
    then have disc_nodes_old_document_h3:
      "h3 ⊢ get_disconnected_nodes old_document →r remove1 child disc_nodes_old_document_h2"
      using h3 old_disc_nodes returns_result_eq set_disconnected_nodes_get_disconnected_nodes
      by fastforce
    have "distinct disc_nodes_old_document_h2"
      using disc_nodes_old_document_h2 local.heap_is_wellformed_disconnected_nodes_distinct wellformed_h2

      by blast


    have "type_wf h2"
    proof (insert h2, induct  parent_opt)
      case None
      then show ?case
        using type_wf by simp
    next
      case (Some option)
      then show ?case
        using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF remove_child_writes]
            type_wf remove_child_types_preserved
        by (simp add: reflp_def transp_def)
    qed
    then have "type_wf h3"
      using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h3]
      using  set_disconnected_nodes_types_preserved
      by(auto simp add: reflp_def transp_def)
    then have "type_wf h'"
      using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h']
      using  set_disconnected_nodes_types_preserved
      by(auto simp add: reflp_def transp_def)

    have disconnected_nodes_eq_h3:
      "⋀doc_ptr disc_nodes. document_ptr ≠ doc_ptr
       ⟹ h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes = h' ⊢ get_disconnected_nodes doc_ptr →r
disc_nodes"
      using get_disconnected_nodes_reads set_disconnected_nodes_writes h'
      apply(rule reads_writes_preserved)
      by (simp add: set_disconnected_nodes_get_disconnected_nodes_different_pointers)
    then have disconnected_nodes_eq2_h3:
      "⋀doc_ptr. document_ptr ≠ doc_ptr
      ⟹ |h3 ⊢ get_disconnected_nodes doc_ptr|r = |h' ⊢ get_disconnected_nodes doc_ptr|r"
      using select_result_eq by force
    have disc_nodes_document_ptr_h2:
      "h2 ⊢ get_disconnected_nodes document_ptr →r disc_nodes_document_ptr_h3"
      using disconnected_nodes_eq_h2 docs_neq disc_nodes_document_ptr_h3 by auto
    have disc_nodes_document_ptr_h': "
```

```
    h' ⊢ get_disconnected_nodes document_ptr →r child # disc_nodes_document_ptr_h3"
    using h' disc_nodes_document_ptr_h3
    using set_disconnected_nodes_get_disconnected_nodes by blast

  have document_ptr_in_heap: "document_ptr |∈| document_ptr_kinds h2"
    using disc_nodes_document_ptr_h3 document_ptr_kinds_eq2_h2 get_disconnected_nodes_ok assms(1)
    unfolding heap_is_wellformed_def
    using disc_nodes_document_ptr_h2 get_disconnected_nodes_ptr_in_heap by blast
  have old_document_in_heap: "old_document |∈| document_ptr_kinds h2"
    using disc_nodes_old_document_h3 document_ptr_kinds_eq2_h2 get_disconnected_nodes_ok assms(1)
    unfolding heap_is_wellformed_def
    using get_disconnected_nodes_ptr_in_heap old_disc_nodes by blast

  have "child ∈ set disc_nodes_old_document_h2"
  proof (insert parent_opt h2, induct parent_opt)
    case None
    then have "h = h2"
      by(auto)
    moreover have "a_owner_document_valid h"
      using assms(1) heap_is_wellformed_def by(simp add: heap_is_wellformed_def)
    ultimately show ?case
      using old_document disc_nodes_old_document_h2 None(1) child_parent_dual[OF assms(1)]
          in_disconnected_nodes_no_parent assms(1) known_ptrs type_wf by blast
  next
    case (Some option)
    then show ?case
      apply(simp split: option.splits)
      using assms(1) disc_nodes_old_document_h2 old_document remove_child_in_disconnected_nodes known_ptrs

      by blast
  qed
  have "child ∉ set (remove1 child disc_nodes_old_document_h2)"
    using disc_nodes_old_document_h3 h3 known_ptrs wellformed_h2 ‹distinct disc_nodes_old_document_h2›

    by auto
  have "child ∉ set disc_nodes_document_ptr_h3"
  proof -
    have "a_distinct_lists h2"
      using heap_is_wellformed_def wellformed_h2 by blast
    then have 0: "distinct (concat (map (λdocument_ptr. |h2 ⊢ get_disconnected_nodes document_ptr|r)

                                                        |h2 ⊢ document_ptr_kinds_M|r))"
      by(simp add: a_distinct_lists_def)
    show ?thesis
      using distinct_concat_map_E(1)[OF 0] ‹child ∈ set disc_nodes_old_document_h2›
          disc_nodes_old_document_h2 disc_nodes_document_ptr_h2
      by (meson ‹type_wf h2› docs_neq known_ptrs local.get_owner_document_disconnected_nodes
            local.known_ptrs_preserved object_ptr_kinds_h_eq3 returns_result_eq wellformed_h2)
  qed

  have child_in_heap: "child |∈| node_ptr_kinds h"
    using get_owner_document_ptr_in_heap[OF is_OK_returns_result_I[OF old_document]]
        node_ptr_kinds_commutes by blast
  have "a_acyclic_heap h2"
    using wellformed_h2 by (simp add: heap_is_wellformed_def)
  have "parent_child_rel h' ⊆ parent_child_rel h2"
  proof
    fix x
    assume "x ∈ parent_child_rel h'"
    then show "x ∈ parent_child_rel h2"
      using object_ptr_kinds_h2_eq3  object_ptr_kinds_h3_eq3 children_eq2_h2 children_eq2_h3
          mem_Collect_eq object_ptr_kinds_M_eq_h3 select_result_eq split_cong
      unfolding parent_child_rel_def
```

```
    by(simp)
qed
then have "a_acyclic_heap h'"
  using ‹a_acyclic_heap h2› acyclic_heap_def acyclic_subset by blast

moreover have "a_all_ptrs_in_heap h2"
  using wellformed_h2 by (simp add: heap_is_wellformed_def)
then have "a_all_ptrs_in_heap h3"
  apply(auto simp add: a_all_ptrs_in_heap_def node_ptr_kinds_eq3_h2 children_eq_h2)[1]
  by (metis (mono_tags, lifting) disc_nodes_old_document_h2 disc_nodes_old_document_h3
           disconnected_nodes_eq_h2 fset_of_list_elem fset_rev_mp returns_result_eq
           set_remove1_subset subsetCE)
then have "a_all_ptrs_in_heap h'"
  apply(auto simp add: a_all_ptrs_in_heap_def node_ptr_kinds_eq3_h3 children_eq_h3)[1]
  by (metis (no_types) NodeMonad.ptr_kinds_ptr_kinds_M child_in_heap disc_nodes_document_ptr_h'
           disc_nodes_document_ptr_h3 disconnected_nodes_eq_h3 fset_mp fset_of_list_elem
           node_ptr_kinds_eq_h node_ptr_kinds_eq_h2 node_ptr_kinds_eq_h3 select_result_I2
           set_ConsD)

moreover have "a_owner_document_valid h2"
  using wellformed_h2 by (simp add: heap_is_wellformed_def)
then have "a_owner_document_valid h'"
  apply(simp add: a_owner_document_valid_def node_ptr_kinds_eq_h2 node_ptr_kinds_eq3_h3
                  object_ptr_kinds_eq_h2 object_ptr_kinds_eq_h3 document_ptr_kinds_eq2_h2
                  document_ptr_kinds_eq2_h3 children_eq2_h2 children_eq2_h3 )
  by (metis (no_types) disc_nodes_document_ptr_h' disc_nodes_document_ptr_h2
                       disc_nodes_old_document_h2 disc_nodes_old_document_h3
                       disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 document_ptr_in_heap
                       document_ptr_kinds_eq3_h2 document_ptr_kinds_eq3_h3 in_set_remove1
                       list.set_intros(1) list.set_intros(2) node_ptr_kinds_eq3_h2
                       node_ptr_kinds_eq3_h3 object_ptr_kinds_h2_eq3 object_ptr_kinds_h3_eq3
                       select_result_I2)

have a_distinct_lists_h2: "a_distinct_lists h2"
  using wellformed_h2 by (simp add: heap_is_wellformed_def)
then have "a_distinct_lists h'"
  apply(auto simp add: a_distinct_lists_def object_ptr_kinds_eq_h3 object_ptr_kinds_eq_h2
                       children_eq2_h2 children_eq2_h3)[1]
proof -
  assume 1: "distinct (concat (map (λptr. |h' ⊢ get_child_nodes ptr|_r)
                       (sorted_list_of_set (fset (object_ptr_kinds h')))))"
    and 2: "distinct (concat (map (λdocument_ptr. |h2 ⊢ get_disconnected_nodes document_ptr|_r)
                                  (sorted_list_of_set (fset (document_ptr_kinds h2)))))"
    and 3: "(⋃x∈fset (object_ptr_kinds h'). set |h' ⊢ get_child_nodes x|_r)
              ∩ (⋃x∈fset (document_ptr_kinds h2). set |h2 ⊢ get_disconnected_nodes x|_r) = {}"
  show "distinct (concat (map (λdocument_ptr. |h' ⊢ get_disconnected_nodes document_ptr|_r)
                                  (sorted_list_of_set (fset (document_ptr_kinds h')))))"
  proof(rule distinct_concat_map_I)
    show "distinct (sorted_list_of_set (fset (document_ptr_kinds h')))"
      by(auto simp add: document_ptr_kinds_M_def )
  next
    fix x
    assume a1: "x ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))"
    have 4: "distinct |h2 ⊢ get_disconnected_nodes x|_r"
      using a_distinct_lists_h2 "2" a1 concat_map_all_distinct document_ptr_kinds_eq2_h2
            document_ptr_kinds_eq2_h3
      by fastforce
    then show "distinct |h' ⊢ get_disconnected_nodes x|_r"
    proof (cases "old_document ≠ x")
      case True
      then show ?thesis
      proof (cases "document_ptr ≠ x")
        case True
```

```
            then show ?thesis
              using disconnected_nodes_eq2_h2[OF ‹old_document ≠ x›]
                    disconnected_nodes_eq2_h3[OF ‹document_ptr ≠ x›] 4
              by(auto)
          next
            case False
            then show ?thesis
              using  disc_nodes_document_ptr_h3 disc_nodes_document_ptr_h' 4
                     ‹child ∉ set disc_nodes_document_ptr_h3›
              by(auto simp add: disconnected_nodes_eq2_h2[OF ‹old_document ≠ x›] )
          qed
        next
          case False
          then show ?thesis
            by (metis (no_types, hide_lams) ‹distinct disc_nodes_old_document_h2›
                      disc_nodes_old_document_h3 disconnected_nodes_eq2_h3
                      distinct_remove1 docs_neq select_result_I2)
        qed
      next
        fix x y
        assume a0: "x ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))"
          and a1: "y ∈ set (sorted_list_of_set (fset (document_ptr_kinds h')))"
          and a2: "x ≠ y"

        moreover have 5: "set |h2 ⊢ get_disconnected_nodes x|_r ∩ set |h2 ⊢ get_disconnected_nodes y|_r
= {}"
          using 2 calculation
          by (auto simp add: document_ptr_kinds_eq3_h2 document_ptr_kinds_eq3_h3 dest: distinct_concat_map_E(1))
        ultimately show "set |h' ⊢ get_disconnected_nodes x|_r ∩ set |h' ⊢ get_disconnected_nodes y|_r =
{}"
        proof(cases "old_document = x")
          case True
          have "old_document ≠ y"
            using ‹x ≠ y› ‹old_document = x› by simp
          have "document_ptr ≠ x"
            using docs_neq ‹old_document = x› by auto
          show ?thesis
          proof(cases "document_ptr = y")
            case True
            then show ?thesis
              using 5 True select_result_I2[OF disc_nodes_document_ptr_h']
                select_result_I2[OF disc_nodes_document_ptr_h2]
                select_result_I2[OF disc_nodes_old_document_h2]
                select_result_I2[OF disc_nodes_old_document_h3] ‹old_document = x›
              by (metis (no_types, lifting) ‹child ∉ set (remove1 child disc_nodes_old_document_h2)›
                        ‹document_ptr ≠ x› disconnected_nodes_eq2_h3 disjoint_iff_not_equal
                        notin_set_remove1 set_ConsD)
          next
            case False
            then show ?thesis
              using 5 select_result_I2[OF disc_nodes_document_ptr_h']
                select_result_I2[OF disc_nodes_document_ptr_h2]
                select_result_I2[OF disc_nodes_old_document_h2]
                select_result_I2[OF disc_nodes_old_document_h3]
                disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 ‹old_document = x›
                docs_neq ‹old_document ≠ y›
              by (metis (no_types, lifting) disjoint_iff_not_equal notin_set_remove1)
          qed
        next
          case False
          then show ?thesis
          proof(cases "old_document = y")
            case True
```

```
    then show ?thesis
    proof(cases "document_ptr = x")
      case True
      show ?thesis
        using 5 select_result_I2[OF disc_nodes_document_ptr_h']
              select_result_I2[OF disc_nodes_document_ptr_h2]
              select_result_I2[OF disc_nodes_old_document_h2]
              select_result_I2[OF disc_nodes_old_document_h3]
              ⟨old_document ≠ x⟩ ⟨old_document = y⟩ ⟨document_ptr = x⟩
        apply(simp)
        by (metis (no_types, lifting) ⟨child ∉ set (remove1 child disc_nodes_old_document_h2)⟩
              disconnected_nodes_eq2_h3 disjoint_iff_not_equal notin_set_remove1)
    next
      case False
      then show ?thesis
        using 5 select_result_I2[OF disc_nodes_document_ptr_h']
              select_result_I2[OF disc_nodes_document_ptr_h2]
              select_result_I2[OF disc_nodes_old_document_h2]
              select_result_I2[OF disc_nodes_old_document_h3]
              ⟨old_document ≠ x⟩ ⟨old_document = y⟩ ⟨document_ptr ≠ x⟩
        by (metis (no_types, lifting) disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3
              disjoint_iff_not_equal docs_neq notin_set_remove1)
    qed
  next
    case False
    have "set |h2 ⊢ get_disconnected_nodes y|_r ∩ set disc_nodes_old_document_h2 = {}"
      by (metis DocumentMonad.ptr_kinds_M_ok DocumentMonad.ptr_kinds_M_ptr_kinds False
              ⟨type_wf h2⟩ a1 disc_nodes_old_document_h2 document_ptr_kinds_M_def
              document_ptr_kinds_eq2_h2 document_ptr_kinds_eq2_h3
              l_ptr_kinds_M.ptr_kinds_ptr_kinds_M local.get_disconnected_nodes_ok
              local.heap_is_wellformed_one_disc_parent returns_result_select_result
              wellformed_h2)
    then show ?thesis
    proof(cases "document_ptr = x")
      case True
      then have "document_ptr ≠ y"
        using ⟨x ≠ y⟩ by auto
      have "set |h2 ⊢ get_disconnected_nodes y|_r ∩ set disc_nodes_old_document_h2 = {}"
        using ⟨set |h2 ⊢ get_disconnected_nodes y|_r ∩ set disc_nodes_old_document_h2 = {}⟩
        by blast
      then show ?thesis
        using 5 select_result_I2[OF disc_nodes_document_ptr_h']
          select_result_I2[OF disc_nodes_document_ptr_h2]
          select_result_I2[OF disc_nodes_old_document_h2]
          select_result_I2[OF disc_nodes_old_document_h3]
          ⟨old_document ≠ x⟩ ⟨old_document ≠ y⟩ ⟨document_ptr = x⟩ ⟨document_ptr ≠ y⟩
          ⟨child ∈ set disc_nodes_old_document_h2⟩ disconnected_nodes_eq2_h2
          disconnected_nodes_eq2_h3
          ⟨set |h2 ⊢ get_disconnected_nodes y|_r ∩ set disc_nodes_old_document_h2 = {}⟩
        by(auto)
    next
      case False
      then show ?thesis
      proof(cases "document_ptr = y")
        case True
        have f1: "set |h2 ⊢ get_disconnected_nodes x|_r ∩ set disc_nodes_document_ptr_h3 = {}"
          using 2 a1 document_ptr_in_heap document_ptr_kinds_eq2_h2 document_ptr_kinds_eq2_h3
              ⟨document_ptr ≠ x⟩ select_result_I2[OF disc_nodes_document_ptr_h3, symmetric]
              disconnected_nodes_eq2_h2[OF docs_neq[symmetric], symmetric]
          by (simp add: "5" True)
        moreover have f1:
            "set |h2 ⊢ get_disconnected_nodes x|_r ∩ set |h2 ⊢ get_disconnected_nodes old_document|_r
= {}"
```

```
            using 2 a1 old_document_in_heap document_ptr_kinds_eq2_h2 document_ptr_kinds_eq2_h3
                ⟨old_document ≠ x⟩
            by (metis (no_types, lifting) a0 distinct_concat_map_E(1) document_ptr_kinds_eq3_h2
                  document_ptr_kinds_eq3_h3 finite_fset fmember.rep_eq set_sorted_list_of_set)
          ultimately show ?thesis
            using 5 select_result_I2[OF disc_nodes_document_ptr_h']
              select_result_I2[OF disc_nodes_old_document_h2] ⟨old_document ≠ x⟩
              ⟨document_ptr ≠ x⟩ ⟨document_ptr = y⟩
              ⟨child ∈ set disc_nodes_old_document_h2⟩ disconnected_nodes_eq2_h2
              disconnected_nodes_eq2_h3
            by auto
        next
          case False
          then show ?thesis
            using 5
              select_result_I2[OF disc_nodes_old_document_h2] ⟨old_document ≠ x⟩
              ⟨document_ptr ≠ x⟩ ⟨document_ptr ≠ y⟩
              ⟨child ∈ set disc_nodes_old_document_h2⟩
              disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3
            by (metis ⟨set |h2 ⊢ get_disconnected_nodes y|_r ∩ set disc_nodes_old_document_h2 = {}⟩

                      empty_iff inf.idem)
        qed
      qed
    qed
  qed
qed
next
  fix x xa xb
  assume 0: "distinct (concat (map (λptr. |h' ⊢ get_child_nodes ptr|_r)
                          (sorted_list_of_set (fset (object_ptr_kinds h')))))"
    and 1: "distinct (concat (map (λdocument_ptr. |h2 ⊢ get_disconnected_nodes document_ptr|_r)
                          (sorted_list_of_set (fset (document_ptr_kinds h2)))))"
    and 2: "(⋃x∈fset (object_ptr_kinds h'). set |h' ⊢ get_child_nodes x|_r)
              ∩ (⋃x∈fset (document_ptr_kinds h2). set |h2 ⊢ get_disconnected_nodes x|_r) = {}"
    and 3: "xa |∈| object_ptr_kinds h'"
    and 4: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
    and 5: "xb |∈| document_ptr_kinds h'"
    and 6: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
  then show False
    using ⟨child ∈ set disc_nodes_old_document_h2⟩ disc_nodes_document_ptr_h'
          disc_nodes_document_ptr_h2 disc_nodes_old_document_h2 disc_nodes_old_document_h3
          disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 document_ptr_kinds_eq2_h2
          document_ptr_kinds_eq2_h3  old_document_in_heap
    apply(auto)[1]
    apply(cases "xb = old_document")
  proof -
    assume a1: "xb = old_document"
    assume a2: "h2 ⊢ get_disconnected_nodes old_document →_r disc_nodes_old_document_h2"
    assume a3: "h3 ⊢ get_disconnected_nodes old_document →_r remove1 child disc_nodes_old_document_h2"
    assume a4: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
    assume "document_ptr_kinds h2 = document_ptr_kinds h'"
    assume a5: "(⋃x∈fset (object_ptr_kinds h'). set |h' ⊢ get_child_nodes x|_r)
              ∩ (⋃x∈fset (document_ptr_kinds h'). set |h2 ⊢ get_disconnected_nodes x|_r) = {}"
    have f6: "old_document |∈| document_ptr_kinds h'"
      using a1 ⟨xb |∈| document_ptr_kinds h'⟩ by blast
    have f7: "|h2 ⊢ get_disconnected_nodes old_document|_r = disc_nodes_old_document_h2"
      using a2 by simp
    have "x ∈ set disc_nodes_old_document_h2"
      using f6 a3 a1 by (metis (no_types) ⟨type_wf h'⟩ ⟨x ∈ set |h' ⊢ get_disconnected_nodes xb|_r⟩

          disconnected_nodes_eq_h3 docs_neq get_disconnected_nodes_ok returns_result_eq
          returns_result_select_result set_remove1_subset subsetCE)
```

```
      then have "set |h' ⊢ get_child_nodes xa|_r ∩  set |h2 ⊢ get_disconnected_nodes xb|_r = {}"
        using f7 f6 a5 a4 ⟨xa |∈| object_ptr_kinds h'⟩
        by fastforce
      then show ?thesis
        using ⟨x ∈ set disc_nodes_old_document_h2⟩ a1 a4 f7 by blast
    next
      assume a1: "xb ≠ old_document"
      assume a2: "h2 ⊢ get_disconnected_nodes document_ptr →_r disc_nodes_document_ptr_h3"
      assume a3: "h2 ⊢ get_disconnected_nodes old_document →_r disc_nodes_old_document_h2"
      assume a4: "xa |∈| object_ptr_kinds h'"
      assume a5: "h' ⊢ get_disconnected_nodes document_ptr →_r child # disc_nodes_document_ptr_h3"
      assume a6: "old_document |∈| document_ptr_kinds h'"
      assume a7: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
      assume a8: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
      assume a9: "document_ptr_kinds h2 = document_ptr_kinds h'"
      assume a10: "⋀doc_ptr. old_document ≠ doc_ptr
            ⟹ |h2 ⊢ get_disconnected_nodes doc_ptr|_r = |h3 ⊢ get_disconnected_nodes doc_ptr|_r"
      assume a11: "⋀doc_ptr. document_ptr ≠ doc_ptr
            ⟹ |h3 ⊢ get_disconnected_nodes doc_ptr|_r = |h' ⊢ get_disconnected_nodes doc_ptr|_r"
      assume a12: "(⋃x∈fset (object_ptr_kinds h'). set |h' ⊢ get_child_nodes x|_r)
                ∩ (⋃x∈fset (document_ptr_kinds h'). set |h2 ⊢ get_disconnected_nodes x|_r) = {}"
      have f13: "⋀d. d ∉ set |h' ⊢ document_ptr_kinds_M|_r ∨ h2 ⊢ ok get_disconnected_nodes d"
        using a9 ⟨type_wf h2⟩ get_disconnected_nodes_ok
        by simp
      then have f14: "|h2 ⊢ get_disconnected_nodes old_document|_r = disc_nodes_old_document_h2"
        using a6 a3 by simp
      have "x ∉ set |h2 ⊢ get_disconnected_nodes xb|_r"
        using a12 a8 a4 ⟨xb |∈| document_ptr_kinds h'⟩
        by (meson UN_I disjoint_iff_not_equal fmember.rep_eq)
      then have "x = child"
        using f13 a11 a10 a7 a5 a2 a1
        by (metis (no_types, lifting) select_result_I2 set_ConsD)
      then have "child ∉ set disc_nodes_old_document_h2"
        using f14 a12 a8 a6 a4
        by (metis ⟨type_wf h'⟩ adopt_node_removes_child assms(1) assms(2) type_wf
                  get_child_nodes_ok known_ptrs local.known_ptrs_known_ptr object_ptr_kinds_h2_eq3
                  object_ptr_kinds_h3_eq3 object_ptr_kinds_h_eq3 returns_result_select_result)
      then show ?thesis
        using ⟨child ∈ set disc_nodes_old_document_h2⟩ by fastforce
    qed
  qed
  ultimately show ?thesis
    using ⟨type_wf h'⟩ ⟨a_owner_document_valid h'⟩ heap_is_wellformed_def by blast
 qed
qed

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 -
  obtain old_document parent_opt h2 where
    old_document: "h ⊢ get_owner_document (cast node_ptr) →_r old_document" and
    parent_opt: "h ⊢ get_parent node_ptr →_r parent_opt" and
    h2: "h ⊢ (case parent_opt of Some parent ⇒ remove_child parent node_ptr | None ⇒ return ()) →_h h2"

    and
    h': "h2 ⊢ (if owner_document ≠ old_document then do {
        old_disc_nodes ← get_disconnected_nodes old_document;
        set_disconnected_nodes old_document (remove1 node_ptr old_disc_nodes);
```

```
        disc_nodes ← get_disconnected_nodes owner_document;
        set_disconnected_nodes owner_document (node_ptr # disc_nodes)
      } else do {
        return ()
      }) →_h h'"
    using assms(2)
    by(auto simp add: adopt_node_def elim!: bind_returns_heap_E
            dest!: pure_returns_heap_eq[rotated, OF get_owner_document_pure]
                   pure_returns_heap_eq[rotated, OF get_parent_pure])

  show ?thesis
  proof (cases "owner_document = old_document")
    case True
    then show ?thesis
    proof (insert parent_opt h2, induct parent_opt)
      case None
      then have "h = h'"
        using h2 h' by(auto)
      then show ?case
        using in_disconnected_nodes_no_parent assms None old_document by blast
    next
      case (Some parent)
      then show ?case
        using remove_child_in_disconnected_nodes known_ptrs True h' assms(3) old_document by auto
    qed
  next
    case False
    then show ?thesis
      using assms(3)  h' list.set_intros(1) select_result_I2 set_disconnected_nodes_get_disconnected_nodes
      apply(auto elim!: bind_returns_heap_E bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure,
rotated])[1]
    proof -
      fix x  and h'a and xb
      assume a1: "h' ⊢ get_disconnected_nodes owner_document →_r disc_nodes"
      assume a2: "⋀h document_ptr disc_nodes h'. h ⊢ set_disconnected_nodes document_ptr disc_nodes →_h
h'
                  ⟹ h' ⊢ get_disconnected_nodes document_ptr →_r disc_nodes"
      assume "h'a ⊢ set_disconnected_nodes owner_document (node_ptr # xb) →_h h'"
      then have "node_ptr # xb = disc_nodes"
        using a2 a1 by (meson returns_result_eq)
      then show ?thesis
        by (meson list.set_intros(1))
    qed
  qed
qed
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
  by(simp add: l_adopt_node_wf_Core_DOM_def instances)
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
  by(simp add: l_adopt_node_wf2_Core_DOM_def instances)
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"

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"
  using heap_is_wellformed_is_l_heap_is_wellformed known_ptrs_is_l_known_ptrs
  apply(auto simp add: l_adopt_node_wf_def l_adopt_node_wf_axioms_def)[1]
  using adopt_node_preserves_wellformedness apply blast
  using adopt_node_removes_child apply blast
  using adopt_node_node_in_disconnected_nodes apply blast
  using adopt_node_removes_first_child apply blast
  done
```

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 -
  obtain owner_document h2 h3 disc_nodes reference_child where
    "h ⊢ (if Some node = child then a_next_sibling node else return child) →_r reference_child" and
    "h ⊢ get_owner_document ptr →_r owner_document" and
    h2: "h ⊢ adopt_node owner_document node →_h h2" and
    "h2 ⊢ get_disconnected_nodes owner_document →_r disc_nodes" and
    h3: "h2 ⊢ set_disconnected_nodes owner_document (remove1 node disc_nodes) →_h h3" and
    h': "h3 ⊢ a_insert_node ptr node reference_child →_h h'"
    using assms(5)
    by(auto simp add: insert_before_def a_ensure_pre_insertion_validity_def
            elim!: bind_returns_heap_E bind_returns_result_E
                   bind_returns_heap_E2[rotated, OF get_child_nodes_pure, rotated]
                   bind_returns_heap_E2[rotated, OF get_parent_pure, rotated]
                   bind_returns_heap_E2[rotated, OF get_ancestors_pure, rotated]
                   bind_returns_heap_E2[rotated, OF get_owner_document_pure, rotated]
```

```
                bind_returns_heap_E2[rotated, OF next_sibling_pure, rotated]
                bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated]
          split: if_splits option.splits)

  have "h2 ⊢ get_child_nodes ptr' →r children"
    using h2 adopt_node_removes_first_child assms(1) assms(2) assms(3) assms(6)
    by simp
  then have "h3 ⊢ get_child_nodes ptr' →r children"
    using h3
    by(auto simp add: set_disconnected_nodes_get_child_nodes
          dest!: reads_writes_separate_forwards[OF get_child_nodes_reads set_disconnected_nodes_writes])
  then show ?thesis
    using h' assms(4)
    apply(auto simp add: a_insert_node_def
            elim!: bind_returns_heap_E bind_returns_heap_E2[rotated, OF get_child_nodes_pure, rotated])[1]
    by(auto simp add: set_child_nodes_get_child_nodes_different_pointers
          elim!: reads_writes_separate_forwards[OF get_child_nodes_reads set_child_nodes_writes])
qed
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
  by(simp add: l_insert_before_wf_Core_DOM_def instances)
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"
  apply(auto simp add: l_insert_before_wf_def l_insert_before_wf_axioms_def instances)[1]
  using insert_before_removes_child apply fast
  done

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
begin
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 -
```

```
  obtain ancestors reference_child owner_document h2 h3 disconnected_nodes_h2 where
    ancestors: "h ⊢ get_ancestors ptr →r ancestors" and
    node_not_in_ancestors: "cast node ∉ set ancestors" and
    reference_child:
       "h ⊢ (if Some node = child then a_next_sibling node else return child) →r reference_child" and
    owner_document: "h ⊢ get_owner_document ptr →r owner_document" and
    h2: "h ⊢ adopt_node owner_document node →h h2" and
    disconnected_nodes_h2: "h2 ⊢ get_disconnected_nodes owner_document →r disconnected_nodes_h2" and
    h3: "h2 ⊢ set_disconnected_nodes owner_document (remove1 node disconnected_nodes_h2) →h h3" and
    h': "h3 ⊢ a_insert_node ptr node reference_child →h h'"
    using assms(2)
    by(auto simp add: insert_before_def a_ensure_pre_insertion_validity_def
            elim!: bind_returns_heap_E bind_returns_result_E
                 bind_returns_heap_E2[rotated, OF get_parent_pure, rotated]
                 bind_returns_heap_E2[rotated, OF get_child_nodes_pure, rotated]
                 bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated]
                 bind_returns_heap_E2[rotated, OF get_ancestors_pure, rotated]
                 bind_returns_heap_E2[rotated, OF next_sibling_pure, rotated]
                 bind_returns_heap_E2[rotated, OF get_owner_document_pure, rotated]
            split: if_splits option.splits)

  have "known_ptr ptr"
    by (meson get_owner_document_ptr_in_heap is_OK_returns_result_I known_ptrs
              l_known_ptrs.known_ptrs_known_ptr l_known_ptrs_axioms owner_document)

  have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF adopt_node_writes h2]
    using type_wf adopt_node_types_preserved
    by(auto simp add: a_remove_child_locs_def reflp_def transp_def)
  then have "type_wf h3"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h3]
    using  set_disconnected_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)
  then show "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF insert_node_writes h']
    using set_child_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  have object_ptr_kinds_M_eq3_h: "object_ptr_kinds h = object_ptr_kinds h2"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF adopt_node_writes h2])
    using adopt_node_pointers_preserved
    apply blast
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_M_eq_h: "⋀ptrs. h ⊢ object_ptr_kinds_M →r ptrs = h2 ⊢ object_ptr_kinds_M
→r ptrs"
    by(simp add: object_ptr_kinds_M_defs )
  then have object_ptr_kinds_M_eq2_h: "|h ⊢ object_ptr_kinds_M|r = |h2 ⊢ object_ptr_kinds_M|r"
    by simp
  then have node_ptr_kinds_eq2_h: "|h ⊢ node_ptr_kinds_M|r = |h2 ⊢ node_ptr_kinds_M|r"
    using node_ptr_kinds_M_eq by blast

  have "known_ptrs h2"
    using known_ptrs object_ptr_kinds_M_eq3_h known_ptrs_preserved by blast

  have wellformed_h2: "heap_is_wellformed h2"
    using adopt_node_preserves_wellformedness[OF wellformed h2] known_ptrs type_wf .

  have object_ptr_kinds_M_eq3_h2: "object_ptr_kinds h2 = object_ptr_kinds h3"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF set_disconnected_nodes_writes h3])
    unfolding a_remove_child_locs_def
```

```
    using set_disconnected_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_M_eq_h2: "⋀ptrs. h2 ⊢ object_ptr_kinds_M →r ptrs = h3 ⊢ object_ptr_kinds_M
→r ptrs"
    by(simp add: object_ptr_kinds_M_defs)
  then have object_ptr_kinds_M_eq2_h2: "|h2 ⊢ object_ptr_kinds_M|r = |h3 ⊢ object_ptr_kinds_M|r"
    by simp
  then have node_ptr_kinds_eq2_h2: "|h2 ⊢ node_ptr_kinds_M|r = |h3 ⊢ node_ptr_kinds_M|r"
    using node_ptr_kinds_M_eq by blast
  have document_ptr_kinds_eq2_h2: "|h2 ⊢ document_ptr_kinds_M|r = |h3 ⊢ document_ptr_kinds_M|r"
    using object_ptr_kinds_M_eq2_h2 document_ptr_kinds_M_eq by auto

  have "known_ptrs h3"
    using object_ptr_kinds_M_eq3_h2 known_ptrs_preserved ‹known_ptrs h2› by blast

  have object_ptr_kinds_M_eq3_h': "object_ptr_kinds h3 = object_ptr_kinds h'"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h = object_ptr_kinds h'",
                                OF insert_node_writes h'])
    unfolding a_remove_child_locs_def
    using set_child_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have object_ptr_kinds_M_eq_h3:
    "⋀ptrs. h3 ⊢ object_ptr_kinds_M →r ptrs = h' ⊢ object_ptr_kinds_M →r ptrs"
    by(simp add: object_ptr_kinds_M_defs)
  then have object_ptr_kinds_M_eq2_h3:
    "|h3 ⊢ object_ptr_kinds_M|r = |h' ⊢ object_ptr_kinds_M|r"
    by simp
  then have node_ptr_kinds_eq2_h3: "|h3 ⊢ node_ptr_kinds_M|r = |h' ⊢ node_ptr_kinds_M|r"
    using node_ptr_kinds_M_eq by blast
  have document_ptr_kinds_eq2_h3: "|h3 ⊢ document_ptr_kinds_M|r = |h' ⊢ document_ptr_kinds_M|r"
    using object_ptr_kinds_M_eq2_h3 document_ptr_kinds_M_eq by auto

  show "known_ptrs h'"
    using object_ptr_kinds_M_eq3_h' known_ptrs_preserved ‹known_ptrs h3› by blast

  have disconnected_nodes_eq_h2:
    "⋀doc_ptr disc_nodes. owner_document ≠ doc_ptr
      ⟹ h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes = h3 ⊢ get_disconnected_nodes doc_ptr →r
disc_nodes"
    using get_disconnected_nodes_reads set_disconnected_nodes_writes h3
    apply(rule reads_writes_preserved)
    by (auto simp add: set_disconnected_nodes_get_disconnected_nodes_different_pointers)
  then have disconnected_nodes_eq2_h2:
     "⋀doc_ptr. doc_ptr ≠ owner_document
     ⟹ |h2 ⊢ get_disconnected_nodes doc_ptr|r = |h3 ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force
  have disconnected_nodes_h3:
   "h3 ⊢ get_disconnected_nodes owner_document →r remove1 node disconnected_nodes_h2"
    using h3 set_disconnected_nodes_get_disconnected_nodes
    by blast

  have disconnected_nodes_eq_h3:
    "⋀doc_ptr disc_nodes. h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                    = h' ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads insert_node_writes  h'
    apply(rule reads_writes_preserved)
    using set_child_nodes_get_disconnected_nodes by fast
  then have disconnected_nodes_eq2_h3:
    "⋀doc_ptr. |h3 ⊢ get_disconnected_nodes doc_ptr|r = |h' ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have children_eq_h2:
     "⋀ptr' children. h2 ⊢ get_child_nodes ptr' →r children = h3 ⊢ get_child_nodes ptr' →r children"
```

```
  using get_child_nodes_reads set_disconnected_nodes_writes h3
  apply(rule reads_writes_preserved)
  by (auto simp add: set_disconnected_nodes_get_child_nodes)
then have children_eq2_h2:
   "⋀ptr'. |h2 ⊢ get_child_nodes ptr'|_r = |h3 ⊢ get_child_nodes ptr'|_r"
  using select_result_eq by force

have children_eq_h3:
  "⋀ptr' children. ptr ≠ ptr'
      ⟹ h3 ⊢ get_child_nodes ptr' →_r children = h' ⊢ get_child_nodes ptr' →_r children"
  using get_child_nodes_reads insert_node_writes h'
  apply(rule reads_writes_preserved)
  by (auto simp add: set_child_nodes_get_child_nodes_different_pointers)
then have children_eq2_h3:
   "⋀ptr'. ptr ≠ ptr' ⟹ |h3 ⊢ get_child_nodes ptr'|_r = |h' ⊢ get_child_nodes ptr'|_r"
  using select_result_eq by force
obtain children_h3 where children_h3: "h3 ⊢ get_child_nodes ptr →_r children_h3"
  using h' a_insert_node_def by auto
have children_h': "h' ⊢ get_child_nodes ptr →_r insert_before_list node reference_child children_h3"
  using h' ‹type_wf h3› ‹known_ptr ptr›
  by(auto simp add: a_insert_node_def elim!: bind_returns_heap_E2
         dest!: set_child_nodes_get_child_nodes returns_result_eq[OF children_h3])

have ptr_in_heap: "ptr |∈| object_ptr_kinds h3"
  using children_h3 get_child_nodes_ptr_in_heap by blast
have node_in_heap: "node |∈| node_ptr_kinds h"
  using h2 adopt_node_child_in_heap by fast
have child_not_in_any_children:
  "⋀p children. h2 ⊢ get_child_nodes p →_r children ⟹ node ∉ set children"
  using wellformed h2 adopt_node_removes_child ‹type_wf h› ‹known_ptrs h› by auto
have "node ∈ set disconnected_nodes_h2"
  using disconnected_nodes_h2 h2 adopt_node_node_in_disconnected_nodes assms(1)
       ‹type_wf h› ‹known_ptrs h› by blast
have node_not_in_disconnected_nodes:
     "⋀d. d |∈| document_ptr_kinds h3 ⟹ node ∉ set |h3 ⊢ get_disconnected_nodes d|_r"
proof -
  fix d
  assume "d |∈| document_ptr_kinds h3"
  show "node ∉ set |h3 ⊢ get_disconnected_nodes d|_r"
  proof (cases "d = owner_document")
    case True
    then show ?thesis
      using disconnected_nodes_h2 wellformed_h2 h3 remove_from_disconnected_nodes_removes
           wellformed_h2 ‹d |∈| document_ptr_kinds h3› disconnected_nodes_h3
      by fastforce
  next
    case False
    then have
      "set |h2 ⊢ get_disconnected_nodes d|_r ∩ set |h2 ⊢ get_disconnected_nodes owner_document|_r = {}"
      using distinct_concat_map_E(1) wellformed_h2
      by (metis (no_types, lifting) ‹d |∈| document_ptr_kinds h3› ‹type_wf h2›
                disconnected_nodes_h2 document_ptr_kinds_M_def document_ptr_kinds_eq2_h2
                l_ptr_kinds_M.ptr_kinds_ptr_kinds_M local.get_disconnected_nodes_ok
                local.heap_is_wellformed_one_disc_parent returns_result_select_result
                select_result_I2)
    then show ?thesis
      using disconnected_nodes_eq2_h2[OF False] ‹node ∈ set disconnected_nodes_h2›
           disconnected_nodes_h2 by fastforce
  qed
qed

have "cast node ≠ ptr"
  using ancestors node_not_in_ancestors get_ancestors_ptr
```

```
    by fast

  obtain ancestors_h2 where ancestors_h2: "h2 ⊢ get_ancestors ptr →r ancestors_h2"
    using get_ancestors_ok object_ptr_kinds_M_eq2_h2 ‹known_ptrs h2› ‹type_wf h2›
    by (metis is_OK_returns_result_E object_ptr_kinds_M_eq3_h2 ptr_in_heap wellformed_h2)
  have ancestors_h3: "h3 ⊢ get_ancestors ptr →r ancestors_h2"
    using get_ancestors_reads set_disconnected_nodes_writes h3
    apply(rule reads_writes_separate_forwards)
    using ‹heap_is_wellformed h2› ancestors_h2
    by (auto simp add: set_disconnected_nodes_get_ancestors)
  have node_not_in_ancestors_h2: "cast node ∉ set ancestors_h2"
    apply(rule get_ancestors_remains_not_in_ancestors[OF assms(1) wellformed_h2 ancestors ancestors_h2])
    using adopt_node_children_subset using h2 ‹known_ptrs h› ‹ type_wf h› apply(blast)
    using node_not_in_ancestors apply(blast)
    using object_ptr_kinds_M_eq3_h apply(blast)
    using ‹known_ptrs h› apply(blast)
    using ‹type_wf h› apply(blast)
    using ‹type_wf h2› by blast

  moreover have "a_acyclic_heap h'"
  proof -
    have "acyclic (parent_child_rel h2)"
      using wellformed_h2  by (simp add: heap_is_wellformed_def acyclic_heap_def)
    then have "acyclic (parent_child_rel h3)"
      by(auto simp add: parent_child_rel_def object_ptr_kinds_M_eq3_h2 children_eq2_h2)
    moreover have "cast node ∉ {x. (x, ptr) ∈ (parent_child_rel h2)*}"
      using get_ancestors_parent_child_rel node_not_in_ancestors_h2 ‹known_ptrs h2› ‹type_wf h2›
      using ancestors_h2 wellformed_h2 by blast
    then have "cast node ∉ {x. (x, ptr) ∈ (parent_child_rel h3)*}"
      by(auto simp add: parent_child_rel_def object_ptr_kinds_M_eq3_h2 children_eq2_h2)
    moreover have "parent_child_rel h' = insert (ptr, cast node) ((parent_child_rel h3))"
      using  children_h3 children_h' ptr_in_heap
      apply(auto simp add: parent_child_rel_def object_ptr_kinds_M_eq3_h' children_eq2_h3
                           insert_before_list_node_in_set)[1]
      apply (metis (no_types, lifting) children_eq2_h3 insert_before_list_in_set select_result_I2)
      by (metis (no_types, lifting) children_eq2_h3 imageI insert_before_list_in_set select_result_I2)
    ultimately show ?thesis
      by(auto simp add: acyclic_heap_def)
  qed


  moreover have "a_all_ptrs_in_heap h2"
    using wellformed_h2  by (simp add: heap_is_wellformed_def)
  have "a_all_ptrs_in_heap h'"
  proof -
    have "a_all_ptrs_in_heap h3"
      using ‹a_all_ptrs_in_heap h2›
      apply(auto simp add: a_all_ptrs_in_heap_def object_ptr_kinds_M_eq2_h2 node_ptr_kinds_eq2_h2
                           children_eq_h2)[1]
      using disconnected_nodes_eq2_h2 disconnected_nodes_h2 disconnected_nodes_h3
      using node_ptr_kinds_eq2_h2 apply auto[1]
      by (metis (no_types, lifting) NodeMonad.ptr_kinds_ptr_kinds_M disconnected_nodes_eq_h2
                disconnected_nodes_h2 disconnected_nodes_h3 fset_mp fset_of_list_subset
                node_ptr_kinds_eq2_h2 select_result_I2 set_remove1_subset)
    have "set children_h3  ⊆ set |h' ⊢ node_ptr_kinds_M|r"
      using children_h3 ‹a_all_ptrs_in_heap h3›
      apply(auto simp add: a_all_ptrs_in_heap_def node_ptr_kinds_eq2_h3)[1]
      by (metis (no_types, hide_lams) fset_mp fset_of_list_elem node_ptr_kinds_commutes object_ptr_kinds_M_eq3_h')
    then have "set (insert_before_list node reference_child children_h3) ⊆ set |h' ⊢ node_ptr_kinds_M|r"
      using node_in_heap
      apply(auto simp add: node_ptr_kinds_eq2_h node_ptr_kinds_eq2_h2 node_ptr_kinds_eq2_h3)[1]
      by (metis (no_types, hide_lams) contra_subsetD finite_set_in insert_before_list_in_set
                node_ptr_kinds_commutes object_ptr_kinds_M_eq3_h object_ptr_kinds_M_eq3_h'
```

```
            object_ptr_kinds_M_eq3_h2)
  then show ?thesis
    using ‹a_all_ptrs_in_heap h3›
    apply(auto simp add: object_ptr_kinds_M_eq3_h' a_all_ptrs_in_heap_def node_ptr_kinds_def
                         node_ptr_kinds_eq2_h3 disconnected_nodes_eq_h3)[1]
    using children_eq_h3 children_h'
    by (metis (no_types, hide_lams) NodeMonad.ptr_kinds_ptr_kinds_M
            ‹set (insert_before_list node reference_child children_h3) ⊆ set |h' ⊢ node_ptr_kinds_M|_r›

            fset.map_comp fset_mp fset_of_list_elem node_ptr_kinds_def node_ptr_kinds_eq2_h3
            returns_result_eq subsetCE)
qed


moreover have "a_distinct_lists h2"
  using wellformed_h2  by (simp add: heap_is_wellformed_def)
then have "a_distinct_lists h3"
proof(auto simp add: a_distinct_lists_def object_ptr_kinds_M_eq2_h2 document_ptr_kinds_eq2_h2
                     children_eq2_h2 intro!: distinct_concat_map_I)[1]
  fix x
  assume 1: "x |∈| document_ptr_kinds h3"
    and 2: "distinct (concat (map (λdocument_ptr. |h2 ⊢ get_disconnected_nodes document_ptr|_r)
                      (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
  show "distinct |h3 ⊢ get_disconnected_nodes x|_r"
    using distinct_concat_map_E(2)[OF 2] select_result_I2[OF disconnected_nodes_h3]
          disconnected_nodes_eq2_h2 select_result_I2[OF disconnected_nodes_h2] 1
    by (metis (full_types) distinct_remove1 finite_fset fmember.rep_eq set_sorted_list_of_set)
next
  fix x y xa
  assume 1: "distinct (concat (map (λdocument_ptr. |h2 ⊢ get_disconnected_nodes document_ptr|_r)
                       (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
    and 2: "x |∈| document_ptr_kinds h3"
    and 3: "y |∈| document_ptr_kinds h3"
    and 4: "x ≠ y"
    and 5: "xa ∈ set |h3 ⊢ get_disconnected_nodes x|_r"
    and 6: "xa ∈ set |h3 ⊢ get_disconnected_nodes y|_r"
  show False
  proof (cases "x = owner_document")
    case True
    then have "y ≠ owner_document"
      using 4 by simp
    show ?thesis
      using distinct_concat_map_E(1)[OF 1]
      using 2 3 4 5 6 select_result_I2[OF disconnected_nodes_h3]  select_result_I2[OF disconnected_nodes_h2]
      apply(auto simp add: True disconnected_nodes_eq2_h2[OF ‹y ≠ owner_document›])[1]
      by (metis (no_types, hide_lams) disconnected_nodes_eq2_h2 disjoint_iff_not_equal notin_set_remove1)
  next
    case False
    then show ?thesis
    proof (cases "y = owner_document")
      case True
      then show ?thesis
      using distinct_concat_map_E(1)[OF 1]
      using 2 3 4 5 6 select_result_I2[OF disconnected_nodes_h3]  select_result_I2[OF disconnected_nodes_h2]
      apply(auto simp add: True disconnected_nodes_eq2_h2[OF ‹x ≠ owner_document›])[1]
      by (metis (no_types, hide_lams) disconnected_nodes_eq2_h2 disjoint_iff_not_equal notin_set_remove1)
    next
      case False
      then show ?thesis
        using distinct_concat_map_E(1)[OF 1, simplified, OF 2 3 4] 5 6
        using disconnected_nodes_eq2_h2 disconnected_nodes_h2 disconnected_nodes_h3
              disjoint_iff_not_equal finite_fset fmember.rep_eq notin_set_remove1 select_result_I2
              set_sorted_list_of_set
```

```
        by (metis (no_types, lifting))
    qed
  qed
next
  fix x xa xb
  assume 1: "(⋃x∈fset (object_ptr_kinds h3). set |h3 ⊢ get_child_nodes x|_r)
                ∩ (⋃x∈fset (document_ptr_kinds h3). set |h2 ⊢ get_disconnected_nodes x|_r) = {}"
    and 2: "xa |∈| object_ptr_kinds h3"
    and 3: "x ∈ set |h3 ⊢ get_child_nodes xa|_r"
    and 4: "xb |∈| document_ptr_kinds h3"
    and 5: "x ∈ set |h3 ⊢ get_disconnected_nodes xb|_r"
  have 6: "set |h3 ⊢ get_child_nodes xa|_r ∩ set |h2 ⊢ get_disconnected_nodes xb|_r = {}"
    using 1 2 4
    by (metis ⟨type_wf h2⟩ children_eq2_h2 document_ptr_kinds_commutes known_ptrs
              local.get_child_nodes_ok local.get_disconnected_nodes_ok
              local.heap_is_wellformed_children_disc_nodes_different local.known_ptrs_known_ptr
              object_ptr_kinds_M_eq3_h object_ptr_kinds_M_eq3_h2 returns_result_select_result
              wellformed_h2)
  show False
  proof (cases "xb = owner_document")
    case True
    then show ?thesis
      using select_result_I2[OF disconnected_nodes_h3,folded select_result_I2[OF disconnected_nodes_h2]]
      by (metis (no_types, lifting) "3" "5" "6" disjoint_iff_not_equal notin_set_remove1)
  next
    case False
    show ?thesis
      using 2 3 4 5 6 unfolding disconnected_nodes_eq2_h2[OF False] by auto
  qed
qed
then have "a_distinct_lists h'"
proof(auto simp add: a_distinct_lists_def document_ptr_kinds_eq2_h3 object_ptr_kinds_M_eq2_h3
                     disconnected_nodes_eq2_h3 intro!: distinct_concat_map_I)[1]
  fix x
  assume 1: "distinct (concat (map (λptr. |h3 ⊢ get_child_nodes ptr|_r)
                                   (sorted_list_of_set (fset (object_ptr_kinds h')))))" and
         2: "x |∈| object_ptr_kinds h'"
  have 3: "⋀p. p |∈| object_ptr_kinds h' ⟹ distinct |h3 ⊢ get_child_nodes p|_r"
    using 1  by (auto elim: distinct_concat_map_E)
  show "distinct |h' ⊢ get_child_nodes x|_r"
  proof(cases "ptr = x")
    case True
    show ?thesis
      using 3[OF 2] children_h3 children_h'
      by(auto simp add: True  insert_before_list_distinct
              dest: child_not_in_any_children[unfolded children_eq_h2])
  next
    case False
    show ?thesis
      using children_eq2_h3[OF False] 3[OF 2] by auto
  qed
next
  fix x y xa
  assume 1: "distinct (concat (map (λptr. |h3 ⊢ get_child_nodes ptr|_r)
                  (sorted_list_of_set (fset (object_ptr_kinds h')))))"
    and 2: "x |∈| object_ptr_kinds h'"
    and 3: "y |∈| object_ptr_kinds h'"
    and 4: "x ≠ y"
    and 5: "xa ∈ set |h' ⊢ get_child_nodes x|_r"
    and 6: "xa ∈ set |h' ⊢ get_child_nodes y|_r"
  have 7:"set |h3 ⊢ get_child_nodes x|_r ∩ set |h3 ⊢ get_child_nodes y|_r = {}"
    using distinct_concat_map_E(1)[OF 1] 2 3 4 by auto
  show False
```

```
  proof (cases "ptr = x")
    case True
    then have "ptr ≠ y"
      using 4 by simp
    then show ?thesis
      using  children_h3 children_h' child_not_in_any_children[unfolded children_eq_h2] 5 6
      apply(auto simp add:  True children_eq2_h3[OF ⟨ptr ≠ y⟩])[1]
      by (metis (no_types, hide_lams) "3" "7" ⟨type_wf h3⟩ children_eq2_h3 disjoint_iff_not_equal
                get_child_nodes_ok insert_before_list_in_set known_ptrs local.known_ptrs_known_ptr
                object_ptr_kinds_M_eq3_h object_ptr_kinds_M_eq3_h'
                object_ptr_kinds_M_eq3_h2 returns_result_select_result select_result_I2)
  next
    case False
    then show ?thesis
    proof (cases "ptr = y")
      case True
      then show ?thesis
        using  children_h3 children_h' child_not_in_any_children[unfolded children_eq_h2] 5 6
        apply(auto simp add:  True children_eq2_h3[OF ⟨ptr ≠ x⟩])[1]
        by (metis (no_types, hide_lams) "2" "4" "7" IntI ⟨known_ptrs h3⟩ ⟨type_wf h'⟩
                  children_eq_h3 empty_iff insert_before_list_in_set local.get_child_nodes_ok
                  local.known_ptrs_known_ptr object_ptr_kinds_M_eq3_h'
                  returns_result_select_result select_result_I2)
      next
      case False
      then show ?thesis
        using children_eq2_h3[OF ⟨ptr ≠ x⟩] children_eq2_h3[OF ⟨ptr ≠ y⟩] 5 6 7 by auto
    qed
  qed
next
  fix x xa xb
  assume 1: " (⋃x∈fset (object_ptr_kinds h'). set |h3 ⊢ get_child_nodes x|_r)
              ∩ (⋃x∈fset (document_ptr_kinds h'). set |h' ⊢ get_disconnected_nodes x|_r) = {} "
    and 2: "xa |∈| object_ptr_kinds h'"
    and 3: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
    and 4: "xb |∈| document_ptr_kinds h'"
    and 5: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
  have 6: "set |h3 ⊢ get_child_nodes xa|_r ∩ set |h' ⊢ get_disconnected_nodes xb|_r = {}"
    using 1 2 3 4 5
  proof -
    have "∀h d. ¬ type_wf h ∨ d |∉| document_ptr_kinds h ∨ h ⊢ ok get_disconnected_nodes d"
      using local.get_disconnected_nodes_ok by satx
    then have "h' ⊢ ok get_disconnected_nodes xb"
      using "4" ⟨type_wf h'⟩ by fastforce
    then have f1: "h3 ⊢ get_disconnected_nodes xb →_r |h' ⊢ get_disconnected_nodes xb|_r"
      by (simp add: disconnected_nodes_eq_h3)
    have "xa |∈| object_ptr_kinds h3"
      using "2" object_ptr_kinds_M_eq3_h' by blast
    then show ?thesis
      using f1 ⟨local.a_distinct_lists h3⟩ local.distinct_lists_no_parent by fastforce
  qed
  show False
  proof (cases "ptr = xa")
    case True
    show ?thesis
      using 6 node_not_in_disconnected_nodes  3 4 5  select_result_I2[OF children_h']
            select_result_I2[OF children_h3] True disconnected_nodes_eq2_h3
      by (metis (no_types, lifting) "2" DocumentMonad.ptr_kinds_ptr_kinds_M
                ⟨a_distinct_lists h3⟩ ⟨type_wf h'⟩ disconnected_nodes_eq_h3
                distinct_lists_no_parent document_ptr_kinds_eq2_h3 get_disconnected_nodes_ok
                insert_before_list_in_set object_ptr_kinds_M_eq3_h' returns_result_select_result)

    next
```

```
      case False
      then show ?thesis
        using 1 2 3 4 5 children_eq2_h3[OF False] by fastforce
    qed
  qed

  moreover have "a_owner_document_valid h2"
    using wellformed_h2  by (simp add: heap_is_wellformed_def)
  then have "a_owner_document_valid h'"
    apply(auto simp add: a_owner_document_valid_def object_ptr_kinds_M_eq2_h2
                         object_ptr_kinds_M_eq2_h3 node_ptr_kinds_eq2_h2 node_ptr_kinds_eq2_h3
                         document_ptr_kinds_eq2_h2 document_ptr_kinds_eq2_h3 children_eq2_h2)[1]
    apply(auto simp add: document_ptr_kinds_eq2_h2[simplified]  document_ptr_kinds_eq2_h3[simplified]
                         object_ptr_kinds_M_eq2_h2[simplified] object_ptr_kinds_M_eq2_h3[simplified]
                         node_ptr_kinds_eq2_h2[simplified] node_ptr_kinds_eq2_h3[simplified])[1]
    apply(auto simp add: disconnected_nodes_eq2_h3[symmetric])[1]
  proof -
    fix node_ptr
    assume 0: "∀node_ptr. node_ptr |∈| node_ptr_kinds h'
              ⟶ (∃document_ptr. document_ptr |∈| document_ptr_kinds h'
                       ∧ node_ptr ∈ set |h2 ⊢ get_disconnected_nodes document_ptr|_r)
                                 ∨ (∃parent_ptr. parent_ptr |∈| object_ptr_kinds h'
                                     ∧ node_ptr ∈ set |h3 ⊢ get_child_nodes parent_ptr|_r)"
      and 1: "node_ptr |∈| node_ptr_kinds h'"
      and 2: "∀parent_ptr. parent_ptr |∈| object_ptr_kinds h'
                                     ⟶ node_ptr ∉ set |h' ⊢ get_child_nodes parent_ptr|_r"
    then have "(∃document_ptr. document_ptr |∈| document_ptr_kinds h'
                                ∧ node_ptr ∈ set |h2 ⊢ get_disconnected_nodes document_ptr|_r)"
      by (metis (no_types, lifting) children_eq2_h3 children_h' children_h3
                                insert_before_list_in_set select_result_I2)
    then show "∃document_ptr. document_ptr |∈| document_ptr_kinds h'
                                  ∧ node_ptr ∈ set |h3 ⊢ get_disconnected_nodes document_ptr|_r"
      by (metis (no_types, hide_lams) "2" children_h' disconnected_nodes_eq2_h2
                                   disconnected_nodes_h2 disconnected_nodes_h3 in_set_remove1
                                   insert_before_list_in_set object_ptr_kinds_M_eq3_h'
                                   ptr_in_heap select_result_I2)
  qed

  ultimately show "heap_is_wellformed h'"
    by (simp add: heap_is_wellformed_def)
qed
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
  by(simp add: l_insert_before_wf2_Core_DOM_def instances)
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"
  apply(auto simp add: l_insert_before_wf2_def l_insert_before_wf2_axioms_def instances)[1]
  using insert_before_heap_is_wellformed_preserved apply(fast, fast, fast)
  done

locale l_append_child_wf_Core_DOM =
  l_adopt_node_Core_DOM +
  l_insert_before_wf_Core_DOM  +
  l_insert_before_wf2  +
  l_append_child_Core_DOM +
  l_get_child_nodes
begin

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 -

  obtain ancestors owner_document h2 h3 disconnected_nodes_h2 where
    ancestors: "h ⊢ get_ancestors ptr →_r ancestors" and
    node_not_in_ancestors: "cast node ∉ set ancestors" and
    owner_document: "h ⊢ get_owner_document ptr →_r owner_document" and
    h2: "h ⊢ adopt_node owner_document node →_h h2" and
    disconnected_nodes_h2: "h2 ⊢ get_disconnected_nodes owner_document →_r disconnected_nodes_h2" and
    h3: "h2 ⊢ set_disconnected_nodes owner_document (remove1 node disconnected_nodes_h2) →_h h3" and
    h': "h3 ⊢ a_insert_node ptr node None →_h h'"
    using assms(5)
    by(auto simp add: append_child_def insert_before_def a_ensure_pre_insertion_validity_def
            elim!: bind_returns_heap_E bind_returns_result_E
         bind_returns_heap_E2[rotated, OF get_parent_pure, rotated]
                bind_returns_heap_E2[rotated, OF get_child_nodes_pure, rotated]
                bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated]
                bind_returns_heap_E2[rotated, OF get_ancestors_pure, rotated]
                bind_returns_heap_E2[rotated, OF next_sibling_pure, rotated]
                bind_returns_heap_E2[rotated, OF get_owner_document_pure, rotated]
          split: if_splits option.splits)

  have "⋀parent. |h ⊢ get_parent node|_r = Some parent ⟹ parent ≠ ptr"
    using assms(1) assms(4) assms(6)
    by (metis (no_types, lifting) assms(2) assms(3) h2 is_OK_returns_heap_I is_OK_returns_result_E
            local.adopt_node_child_in_heap local.get_parent_child_dual local.get_parent_ok
            select_result_I2)
  have "h2 ⊢ get_child_nodes ptr →_r xs"
    using get_child_nodes_reads adopt_node_writes h2 assms(4)
    apply(rule reads_writes_separate_forwards)
    using ⟨⋀parent. |h ⊢ get_parent node|_r = Some parent ⟹ parent ≠ ptr⟩
    apply(auto simp add: adopt_node_locs_def remove_child_locs_def)[1]
    by (meson local.set_child_nodes_get_child_nodes_different_pointers)

  have "h3 ⊢ get_child_nodes ptr →_r xs"
    using get_child_nodes_reads set_disconnected_nodes_writes h3 ⟨h2 ⊢ get_child_nodes ptr →_r xs⟩
    apply(rule reads_writes_separate_forwards)
    by(auto)

  have "ptr |∈| object_ptr_kinds h"
```

```
    by (meson ancestors is_OK_returns_result_I local.get_ancestors_ptr_in_heap)
  then
  have "known_ptr ptr"
    using assms(3)
    using local.known_ptrs_known_ptr by blast

  have "type_wf h2"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF adopt_node_writes h2]
    using adopt_node_types_preserved ‹type_wf h›
    by(auto simp add: adopt_node_locs_def remove_child_locs_def reflp_def transp_def split: if_splits)
  then
  have "type_wf h3"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h3]
    using  set_disconnected_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  show "h' ⊢ get_child_nodes ptr →r xs@[node]"
    using h'
    apply(auto simp add: a_insert_node_def
           dest!: bind_returns_heap_E3[rotated, OF ‹h3 ⊢ get_child_nodes ptr →r xs›
                  get_child_nodes_pure, rotated])[1]
    using ‹type_wf h3› set_child_nodes_get_child_nodes ‹known_ptr ptr›
    by metis
qed

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"
  using assms
  apply(induct nodes arbitrary: h xs)
  apply(simp)
proof(auto elim!: bind_returns_heap_E)[1] fix a nodes h xs h'a
  assume 0: "(⋀h xs. heap_is_wellformed h ⟹ type_wf h ⟹ known_ptrs h
             ⟹ h ⊢ get_child_nodes ptr →r xs ⟹ h ⊢ forall_M (append_child ptr) nodes →h h'
             ⟹ set nodes ∩ set xs = {} ⟹ h' ⊢ get_child_nodes ptr →r xs @ nodes)"
    and 1: "heap_is_wellformed h"
    and 2: "type_wf h"
    and 3: "known_ptrs h"
    and 4: "h ⊢ get_child_nodes ptr →r xs"
    and 5: "h ⊢ append_child ptr a →r ()"
    and 6: "h ⊢ append_child ptr a →h h'a"
    and 7: "h'a ⊢ forall_M (append_child ptr) nodes →h h'"
    and 8: "a ∉ set xs"
    and 9: "set nodes ∩ set xs = {}"
    and 10: "a ∉ set nodes"
    and 11: "distinct nodes"
  then have "h'a ⊢ get_child_nodes ptr →r xs @ [a]"
    using append_child_children 6
    using "1" "2" "3" "4" "8" by blast

  moreover have "heap_is_wellformed h'a" and "type_wf h'a" and "known_ptrs h'a"
    using insert_before_heap_is_wellformed_preserved insert_before_preserves_known_ptrs
          insert_before_preserves_type_wf 1 2 3 6 append_child_def
    by metis+
  moreover have "set nodes ∩ set (xs @ [a]) = {}"
    using 9 10
    by auto
  ultimately show "h' ⊢ get_child_nodes ptr →r xs @ a # nodes"
```

```
    using 0 7
    by fastforce
qed


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"
  using assms append_child_for_all_on_children
  by force
end

interpretation i_append_child_wf?: l_append_child_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 get_ancestors get_ancestors_locs
                                   insert_before insert_before_locs append_child heap_is_wellformed
                                   parent_child_rel
  by(auto simp add: l_append_child_wf_Core_DOM_def instances)
```

6.3.12 create_element

```
locale l_create_element_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_element_get_disconnected_nodes get_disconnected_nodes get_disconnected_nodes_locs +
  l_set_tag_type_get_disconnected_nodes type_wf set_tag_type set_tag_type_locs
                                get_disconnected_nodes get_disconnected_nodes_locs +
  l_create_element_Core_DOM get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
                            set_disconnected_nodes_locs set_tag_type set_tag_type_locs create_element +
  l_new_element_get_child_nodes type_wf known_ptr get_child_nodes get_child_nodes_locs +
  l_set_tag_type_get_child_nodes type_wf set_tag_type set_tag_type_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_type :: "(_) element_ptr ⇒ char list ⇒ ((_) heap, exception, unit) prog"
  and set_tag_type_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'"
proof -
  obtain new_element_ptr h2 h3 disc_nodes_h3 where
    new_element_ptr: "h ⊢ new_element →_r new_element_ptr" and
    h2: "h ⊢ new_element →_h h2" and
    h3: "h2 ⊢ set_tag_type new_element_ptr tag →_h h3" and
    disc_nodes_h3: "h3 ⊢ get_disconnected_nodes document_ptr →_r disc_nodes_h3" and
    h': "h3 ⊢ set_disconnected_nodes document_ptr (cast new_element_ptr # disc_nodes_h3) →_h h'"
    using assms(2)
    by(auto simp add: create_element_def
            elim!: bind_returns_heap_E
                   bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated] )

  have "new_element_ptr ∉ set |h ⊢ element_ptr_kinds_M|_r"
    using new_element_ptr ElementMonad.ptr_kinds_ptr_kinds_M h2
    using new_element_ptr_not_in_heap by blast
  then have "cast new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r"
    by simp
  then have "cast new_element_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r"
    by simp

  have object_ptr_kinds_eq_h: "object_ptr_kinds h2 = object_ptr_kinds h |∪| {|cast new_element_ptr|}"
    using new_element_new_ptr h2 new_element_ptr by blast
  then have node_ptr_kinds_eq_h: "node_ptr_kinds h2 = node_ptr_kinds h |∪| {|cast new_element_ptr|}"
    apply(simp add: node_ptr_kinds_def)
    by force
  then have element_ptr_kinds_eq_h: "element_ptr_kinds h2 = element_ptr_kinds h |∪| {|new_element_ptr|}"
    apply(simp add: element_ptr_kinds_def)
    by force
  have character_data_ptr_kinds_eq_h: "character_data_ptr_kinds h2 = character_data_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: node_ptr_kinds_def character_data_ptr_kinds_def)
  have document_ptr_kinds_eq_h: "document_ptr_kinds h2 = document_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: document_ptr_kinds_def)

  have object_ptr_kinds_eq_h2: "object_ptr_kinds h3 = object_ptr_kinds h2"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h", OF set_tag_type_writes
h3])
    using set_tag_type_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h2: "document_ptr_kinds h3 = document_ptr_kinds h2"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h2: "node_ptr_kinds h3 = node_ptr_kinds h2"
    using object_ptr_kinds_eq_h2
    by(auto simp add: node_ptr_kinds_def)

  have object_ptr_kinds_eq_h3: "object_ptr_kinds h' = object_ptr_kinds h3"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h",
                                OF set_disconnected_nodes_writes h'])
    using set_disconnected_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h3: "document_ptr_kinds h' = document_ptr_kinds h3"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h3: "node_ptr_kinds h' = node_ptr_kinds h3"
    using object_ptr_kinds_eq_h3
    by(auto simp add: node_ptr_kinds_def)
```

```
  have "document_ptr |∈| document_ptr_kinds h"
    using disc_nodes_h3 document_ptr_kinds_eq_h object_ptr_kinds_eq_h2
          get_disconnected_nodes_ptr_in_heap ‹type_wf h› document_ptr_kinds_def
    by (metis is_OK_returns_result_I)

  have children_eq_h: "⋀(ptr'::(_) object_ptr) children. ptr' ≠ cast new_element_ptr
            ⟹ h ⊢ get_child_nodes ptr' →r children = h2 ⊢ get_child_nodes ptr' →r children"
    using get_child_nodes_reads h2 get_child_nodes_new_element[rotated, OF new_element_ptr h2]
    apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
    by blast+
  then have children_eq2_h: "⋀ptr'. ptr' ≠ cast new_element_ptr
                                  ⟹ |h ⊢ get_child_nodes ptr'|r = |h2 ⊢ get_child_nodes ptr'|r"
    using select_result_eq by force

  have "h2 ⊢ get_child_nodes (cast new_element_ptr) →r []"
    using new_element_ptr h2 new_element_ptr_in_heap[OF h2 new_element_ptr]
          new_element_is_element_ptr[OF new_element_ptr] new_element_no_child_nodes
    by blast
  have disconnected_nodes_eq_h:
    "⋀doc_ptr disc_nodes. h ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                        = h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads h2 get_disconnected_nodes_new_element[OF new_element_ptr h2]
    apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
    by blast+
  then have disconnected_nodes_eq2_h:
    "⋀doc_ptr. |h ⊢ get_disconnected_nodes doc_ptr|r = |h2 ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have children_eq_h2:
    "⋀ptr' children. h2 ⊢ get_child_nodes ptr' →r children = h3 ⊢ get_child_nodes ptr' →r children"
    using get_child_nodes_reads set_tag_type_writes h3
    apply(rule reads_writes_preserved)
    by(auto simp add: set_tag_type_get_child_nodes)
  then have children_eq2_h2: "⋀ptr'. |h2 ⊢ get_child_nodes ptr'|r = |h3 ⊢ get_child_nodes ptr'|r"
    using select_result_eq by force
  have disconnected_nodes_eq_h2:
    "⋀doc_ptr disc_nodes. h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                          = h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads set_tag_type_writes h3
    apply(rule reads_writes_preserved)
    by(auto simp add: set_tag_type_get_disconnected_nodes)
  then have disconnected_nodes_eq2_h2:
    "⋀doc_ptr. |h2 ⊢ get_disconnected_nodes doc_ptr|r = |h3 ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have "type_wf h2"
    using ‹type_wf h› new_element_types_preserved h2 by blast
  then have "type_wf h3"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_tag_type_writes h3]
    using  set_tag_type_types_preserved
    by(auto simp add: reflp_def transp_def)
  then have "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h']
    using  set_disconnected_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  have children_eq_h3:
    "⋀ptr' children. h3 ⊢ get_child_nodes ptr' →r children = h' ⊢ get_child_nodes ptr' →r children"
    using get_child_nodes_reads set_disconnected_nodes_writes h'
    apply(rule reads_writes_preserved)
    by(auto simp add: set_disconnected_nodes_get_child_nodes)
  then have children_eq2_h3: "⋀ptr'. |h3 ⊢ get_child_nodes ptr'|r = |h' ⊢ get_child_nodes ptr'|r"
```

```
  using select_result_eq by force
have disconnected_nodes_eq_h3:
  "⋀doc_ptr disc_nodes. document_ptr ≠ doc_ptr
     ⟹ h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                          = h' ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
  using get_disconnected_nodes_reads set_disconnected_nodes_writes h'
  apply(rule reads_writes_preserved)
  by(auto simp add: set_disconnected_nodes_get_disconnected_nodes_different_pointers)
then have disconnected_nodes_eq2_h3:
  "⋀doc_ptr. document_ptr ≠ doc_ptr
            ⟹ |h3 ⊢ get_disconnected_nodes doc_ptr|r = |h' ⊢ get_disconnected_nodes doc_ptr|r"
  using select_result_eq by force

have disc_nodes_document_ptr_h2: "h2 ⊢ get_disconnected_nodes document_ptr →r disc_nodes_h3"
  using disconnected_nodes_eq_h2 disc_nodes_h3 by auto
then have disc_nodes_document_ptr_h: "h ⊢ get_disconnected_nodes document_ptr →r disc_nodes_h3"
  using disconnected_nodes_eq_h by auto
then have "cast new_element_ptr ∉ set disc_nodes_h3"
  using ‹heap_is_wellformed h›
  using ‹cast_element_ptr2node_ptr new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|r›
        a_all_ptrs_in_heap_def heap_is_wellformed_def
  by (meson NodeMonad.ptr_kinds_ptr_kinds_M fset_mp fset_of_list_elem )

have "acyclic (parent_child_rel h)"
  using ‹heap_is_wellformed h›
  by (simp add: heap_is_wellformed_def acyclic_heap_def)
also have "parent_child_rel h = parent_child_rel h2"
proof(auto simp add: parent_child_rel_def)[1]
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|r"
  then show "a |∈| object_ptr_kinds h2"
    by (simp add: object_ptr_kinds_eq_h)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|r"
  then show "x ∈ set |h2 ⊢ get_child_nodes a|r"
    by (metis ObjectMonad.ptr_kinds_ptr_kinds_M
          ‹cast_element_ptr2object_ptr new_element_ptr ∉ set |h ⊢ object_ptr_kinds_M|r› children_eq2_h)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h2"
     and 1: "x ∈ set |h2 ⊢ get_child_nodes a|r"
  then show "a |∈| object_ptr_kinds h"
    using object_ptr_kinds_eq_h ‹h2 ⊢ get_child_nodes (cast_element_ptr2object_ptr new_element_ptr) →r []›
    by(auto)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h2"
    and 1: "x ∈ set |h2 ⊢ get_child_nodes a|r"
  then show "x ∈ set |h ⊢ get_child_nodes a|r"
    by (metis (no_types, lifting)
                          ‹h2 ⊢ get_child_nodes (cast_element_ptr2object_ptr new_element_ptr) →r []›
                          children_eq2_h empty_iff empty_set image_eqI select_result_I2)
qed
also have "... = parent_child_rel h3"
  by(auto simp add: parent_child_rel_def object_ptr_kinds_eq_h2 children_eq2_h2)
also have "... = parent_child_rel h'"
  by(auto simp add: parent_child_rel_def object_ptr_kinds_eq_h3 children_eq2_h3)
finally have "a_acyclic_heap h'"
  by (simp add: acyclic_heap_def)
```

```
  have "a_all_ptrs_in_heap h"
    using ‹heap_is_wellformed h› by (simp add: heap_is_wellformed_def)
  then have "a_all_ptrs_in_heap h2"
    apply(auto simp add: a_all_ptrs_in_heap_def)[1]
    using node_ptr_kinds_eq_h
          ‹cast new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r›
          ‹h2 ⊢ get_child_nodes (cast new_element_ptr) →_r []›
     apply (metis (no_types, hide_lams) children_eq_h fempty_iff fset_mp fset_of_list_simps(1)
                  funionCI select_result_I2)
    by (simp add: disconnected_nodes_eq_h fset_rev_mp node_ptr_kinds_eq_h)
  then have "a_all_ptrs_in_heap h3"
    by(auto simp add: a_all_ptrs_in_heap_def object_ptr_kinds_eq_h2 node_ptr_kinds_def
                      children_eq_h2 disconnected_nodes_eq_h2)
  then have "a_all_ptrs_in_heap h'"
    apply(auto simp add: a_all_ptrs_in_heap_def object_ptr_kinds_eq_h3 node_ptr_kinds_def children_eq_h3
)[1]
    using disconnected_nodes_eq_h3 object_ptr_kinds_eq_h object_ptr_kinds_eq_h2
    by (metis (no_types, lifting) disc_nodes_h3 finsertCI fset.map_comp fset_mp fset_of_list_elem
              funion_finsert_right h' local.set_disconnected_nodes_get_disconnected_nodes
              node_ptr_kinds_def node_ptr_kinds_eq_h select_result_I2 set_ConsD)

  have "⋀p. p |∈| object_ptr_kinds h ⟹ cast new_element_ptr ∉ set |h ⊢ get_child_nodes p|_r"
    using ‹heap_is_wellformed h› ‹cast_element_ptr2node_ptr new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r›
          heap_is_wellformed_children_in_heap
    by (meson NodeMonad.ptr_kinds_ptr_kinds_M a_all_ptrs_in_heap_def assms(3) assms(4) fset_mp
              fset_of_list_elem get_child_nodes_ok known_ptrs_known_ptr returns_result_select_result)
  then have "⋀p. p |∈| object_ptr_kinds h2 ⟹ cast new_element_ptr ∉ set |h2 ⊢ get_child_nodes p|_r"
    using children_eq2_h
    apply(auto simp add: object_ptr_kinds_eq_h)[1]
    using ‹h2 ⊢ get_child_nodes (cast_element_ptr2object_ptr new_element_ptr) →_r []› apply auto[1]
    by (metis ObjectMonad.ptr_kinds_ptr_kinds_M
              ‹cast_element_ptr2object_ptr new_element_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r›)
  then have "⋀p. p |∈| object_ptr_kinds h3 ⟹ cast new_element_ptr ∉ set |h3 ⊢ get_child_nodes p|_r"
    using object_ptr_kinds_eq_h2 children_eq2_h2 by auto
  then have new_element_ptr_not_in_any_children:
    "⋀p. p |∈| object_ptr_kinds h' ⟹ cast new_element_ptr ∉ set |h' ⊢ get_child_nodes p|_r"
    using object_ptr_kinds_eq_h3 children_eq2_h3 by auto

  have "a_distinct_lists h"
    using ‹heap_is_wellformed h›
    by (simp add: heap_is_wellformed_def)
  then have "a_distinct_lists h2"

    using ‹h2 ⊢ get_child_nodes (cast new_element_ptr) →_r []›
    apply(auto simp add: a_distinct_lists_def object_ptr_kinds_eq_h document_ptr_kinds_eq_h
        disconnected_nodes_eq2_h intro!: distinct_concat_map_I)[1]
       apply (metis distinct_sorted_list_of_set finite_fset sorted_list_of_set_insert)
      apply(case_tac "x=cast new_element_ptr")
       apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
      apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
     apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
     apply (metis IntI assms(1) assms(3) assms(4) empty_iff local.get_child_nodes_ok
        local.heap_is_wellformed_one_parent local.known_ptrs_known_ptr returns_result_select_result)
    apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
    by (metis ‹local.a_distinct_lists h› ‹type_wf h2› disconnected_nodes_eq_h document_ptr_kinds_eq_h
        local.distinct_lists_no_parent local.get_disconnected_nodes_ok returns_result_select_result)

  then have "a_distinct_lists h3"
    by(auto simp add: a_distinct_lists_def disconnected_nodes_eq2_h2 document_ptr_kinds_eq_h2
                      children_eq2_h2 object_ptr_kinds_eq_h2)
  then have "a_distinct_lists h'"
  proof(auto simp add: a_distinct_lists_def disconnected_nodes_eq2_h3 children_eq2_h3
                       object_ptr_kinds_eq_h3 document_ptr_kinds_eq_h3
```

```
          intro!: distinct_concat_map_I)[1]
  fix x
  assume "distinct (concat (map (λdocument_ptr. |h3 ⊢ get_disconnected_nodes document_ptr|_r)
                                        (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
    and "x |∈| document_ptr_kinds h3"
  then show "distinct |h' ⊢ get_disconnected_nodes x|_r"
    using document_ptr_kinds_eq_h3 disconnected_nodes_eq_h3 h' set_disconnected_nodes_get_disconnected_nodes
    by (metis (no_types, lifting) ⟨cast_element_ptr2node_ptr new_element_ptr ∉ set disc_nodes_h3⟩
          ⟨a_distinct_lists h3⟩ ⟨type_wf h'⟩ disc_nodes_h3 distinct.simps(2)
          distinct_lists_disconnected_nodes get_disconnected_nodes_ok returns_result_eq
          returns_result_select_result)
next
  fix x y xa
  assume "distinct (concat (map (λdocument_ptr. |h3 ⊢ get_disconnected_nodes document_ptr|_r)
                                      (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
    and "x |∈| document_ptr_kinds h3"
    and "y |∈| document_ptr_kinds h3"
    and "x ≠ y"
    and "xa ∈ set |h' ⊢ get_disconnected_nodes x|_r"
    and "xa ∈ set |h' ⊢ get_disconnected_nodes y|_r"
  moreover have "set |h3 ⊢ get_disconnected_nodes x|_r ∩ set |h3 ⊢ get_disconnected_nodes y|_r = {}"
    using calculation by(auto dest: distinct_concat_map_E(1))
  ultimately show "False"
    apply(-)
    apply(cases "x = document_ptr")
     apply (metis (no_types) NodeMonad.ptr_kinds_ptr_kinds_M
                ⟨a_all_ptrs_in_heap h⟩
                ⟨cast_element_ptr2node_ptr new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r⟩
                a_all_ptrs_in_heap_def assms(3) disc_nodes_h3 disconnected_nodes_eq2_h
                disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 disjoint_iff_not_equal
                document_ptr_kinds_eq_h document_ptr_kinds_eq_h2 fset_mp fset_of_list_elem
                get_disconnected_nodes_ok h' returns_result_select_result select_result_I2
                set_ConsD set_disconnected_nodes_get_disconnected_nodes)
    apply(cases "y = document_ptr" )
    apply (metis (no_types) NodeMonad.ptr_kinds_ptr_kinds_M
               ⟨a_all_ptrs_in_heap h⟩
               ⟨cast_element_ptr2node_ptr new_element_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r⟩
               a_all_ptrs_in_heap_def assms(3) disc_nodes_h3 disconnected_nodes_eq2_h
               disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 disjoint_iff_not_equal
               document_ptr_kinds_eq_h document_ptr_kinds_eq_h2 fset_mp fset_of_list_elem
               get_disconnected_nodes_ok h' returns_result_select_result select_result_I2
               set_ConsD set_disconnected_nodes_get_disconnected_nodes)
    using disconnected_nodes_eq2_h3 by auto
next
  fix x xa xb
  assume 2: "(⋃x∈fset (object_ptr_kinds h3). set |h' ⊢ get_child_nodes x|_r)
               ∩ (⋃x∈fset (document_ptr_kinds h3). set |h3 ⊢ get_disconnected_nodes x|_r) = {}"
    and 3: "xa |∈| object_ptr_kinds h3"
    and 4: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
    and 5: "xb |∈| document_ptr_kinds h3"
    and 6: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
   show "False"
    using disc_nodes_document_ptr_h disconnected_nodes_eq2_h3
    apply -
    apply(cases "xb = document_ptr")
     apply (metis (no_types, hide_lams) "3" "4" "6"
                ⟨⋀p. p |∈| object_ptr_kinds h3
                  ⟹ cast_element_ptr2node_ptr new_element_ptr ∉ set |h3 ⊢ get_child_nodes p|_r⟩
               ⟨a_distinct_lists h3⟩ children_eq2_h3 disc_nodes_h3 distinct_lists_no_parent h'
               select_result_I2 set_ConsD set_disconnected_nodes_get_disconnected_nodes)
    by (metis "3" "4" "5" "6" ⟨a_distinct_lists h3⟩ ⟨type_wf h3⟩ children_eq2_h3
          distinct_lists_no_parent get_disconnected_nodes_ok returns_result_select_result)
qed
```

```
  have "a_owner_document_valid h"
    using ⟨heap_is_wellformed h⟩ by (simp add: heap_is_wellformed_def)
  then have "a_owner_document_valid h'"
    using disc_nodes_h3 ⟨document_ptr |∈| document_ptr_kinds h⟩
    apply(auto simp add: a_owner_document_valid_def)[1]
    apply(auto simp add:  object_ptr_kinds_eq_h object_ptr_kinds_eq_h3 )[1]
    apply(auto simp add: object_ptr_kinds_eq_h2)[1]
    apply(auto simp add:  document_ptr_kinds_eq_h document_ptr_kinds_eq_h3 )[1]
    apply(auto simp add: document_ptr_kinds_eq_h2)[1]
    apply(auto simp add:  node_ptr_kinds_eq_h node_ptr_kinds_eq_h3 )[1]
    apply(auto simp add: node_ptr_kinds_eq_h2 node_ptr_kinds_eq_h )[1]
     apply(auto simp add: children_eq2_h2[symmetric] children_eq2_h3[symmetric]
        disconnected_nodes_eq2_h disconnected_nodes_eq2_h2
        disconnected_nodes_eq2_h3)[1]
     apply (metis (no_types, lifting) document_ptr_kinds_eq_h h' list.set_intros(1)
        local.set_disconnected_nodes_get_disconnected_nodes select_result_I2)
    apply(simp add: object_ptr_kinds_eq_h)
  proof -
    fix node_ptr :: "(_) node_ptr"
    assume a1: "∀node_ptr. node_ptr |∈| node_ptr_kinds h ⟶ (∃document_ptr. document_ptr |∈| document_ptr_kind
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)"
    assume a2: "node_ptr |∈| node_ptr_kinds h"
    assume a3: "∀parent_ptr. (parent_ptr = cast_element_ptr2object_ptr new_element_ptr ⟶ node_ptr ∉ set
|h' ⊢ get_child_nodes (cast_element_ptr2object_ptr new_element_ptr)|_r) ∧ (parent_ptr |∈| object_ptr_kinds h
⟶ node_ptr ∉ set |h' ⊢ get_child_nodes parent_ptr|_r)"
    assume a4: "document_ptr |∈| document_ptr_kinds h"
    assume a5: "h3 ⊢ get_disconnected_nodes document_ptr →_r disc_nodes_h3"
    obtain dd :: "(_) node_ptr ⇒ (_) document_ptr" where
      "∀x0. (∃v1. v1 |∈| document_ptr_kinds h ∧ x0 ∈ set |h ⊢ get_disconnected_nodes v1|_r) = (dd x0 |∈|
document_ptr_kinds h ∧ x0 ∈ set |h ⊢ get_disconnected_nodes (dd x0)|_r)"
      by moura
    then have f6: "dd node_ptr |∈| document_ptr_kinds h ∧ node_ptr ∈ set |h ⊢ get_disconnected_nodes
(dd node_ptr)|_r"
      using a3 a2 a1 by (metis (no_types) ⟨cast_element_ptr2object_ptr new_element_ptr ∉ set |h ⊢ object_ptr_kinds_M|
children_eq2_h children_eq2_h2 children_eq2_h3 l_ptr_kinds_M.ptr_kinds_ptr_kinds_M object_ptr_kinds_M_def)
    moreover
    { assume "|h ⊢ get_disconnected_nodes (dd node_ptr)|_r ≠ disc_nodes_h3"
      then have "document_ptr ≠ dd node_ptr"
        using a5 disconnected_nodes_eq2_h disconnected_nodes_eq2_h2 by force
      then have "∃d. d |∈| document_ptr_kinds h2 ∧ node_ptr ∈ set |h' ⊢ get_disconnected_nodes d|_r"
        using f6 disconnected_nodes_eq2_h disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 document_ptr_kinds_eq
by auto }
    ultimately show "∃d. d |∈| document_ptr_kinds h2 ∧ node_ptr ∈ set |h' ⊢ get_disconnected_nodes d|_r"
      using a4 by (metis (no_types) document_ptr_kinds_eq_h h' insert_iff list.set(2) local.set_disconnected_nodes
select_result_I2)
  qed
  show "heap_is_wellformed h'"
    using ⟨a_acyclic_heap h'⟩ ⟨a_all_ptrs_in_heap h'⟩ ⟨a_distinct_lists h'⟩ ⟨a_owner_document_valid h'⟩
    by(simp add: heap_is_wellformed_def)
qed
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_type set_tag_type_locs
    set_disconnected_nodes set_disconnected_nodes_locs create_element
  using instances
  by(auto simp add: l_create_element_wf_Core_DOM_def)
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
    set_val set_val_locs get_disconnected_nodes get_disconnected_nodes_locs set_disconnected_nodes
    set_disconnected_nodes_locs create_character_data
  + 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 ⇒ (('a) heap, exception, ('b) node_ptr list) prog"
  and get_child_nodes_locs :: "(') object_ptr ⇒ (('a) heap ⇒ ('b) heap ⇒ bool) set"
  and get_disconnected_nodes :: "(') document_ptr ⇒ (('a) heap, exception, ('b) node_ptr list) prog"
  and get_disconnected_nodes_locs :: "(') document_ptr ⇒ (('a) heap ⇒ ('b) heap ⇒ bool) set"
  and heap_is_wellformed :: "('a) heap ⇒ bool"
  and parent_child_rel :: "('a) heap ⇒ (('b) object_ptr × ('c) object_ptr) set"
  and set_val :: "(') character_data_ptr ⇒ char list ⇒ (('a) heap, exception, unit) prog"
  and set_val_locs :: "(') character_data_ptr ⇒ (('a) heap, exception, unit) prog set"
  and set_disconnected_nodes ::
      "(') document_ptr ⇒ ('b) node_ptr list ⇒ (('a) heap, exception, unit) prog"
  and set_disconnected_nodes_locs :: "(') document_ptr ⇒ (('a) heap, exception, unit) prog set"
  and create_character_data ::
      "(') document_ptr ⇒ char list ⇒ (('a) heap, exception, ('b) character_data_ptr) prog"
  and known_ptrs :: "('a) 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'"
proof -
  obtain new_character_data_ptr h2 h3 disc_nodes_h3 where
    new_character_data_ptr: "h ⊢ new_character_data →_r new_character_data_ptr" and
    h2: "h ⊢ new_character_data →_h h2" and
    h3: "h2 ⊢ set_val new_character_data_ptr text →_h h3" and
    disc_nodes_h3: "h3 ⊢ get_disconnected_nodes document_ptr →_r disc_nodes_h3" and
    h': "h3 ⊢ set_disconnected_nodes document_ptr (cast new_character_data_ptr # disc_nodes_h3) →_h h'"
    using assms(2)
    by(auto simp add: create_character_data_def
            elim!: bind_returns_heap_E
                   bind_returns_heap_E2[rotated, OF get_disconnected_nodes_pure, rotated] )
```

```
  have "new_character_data_ptr ∉ set |h ⊢ character_data_ptr_kinds_M|_r"
    using new_character_data_ptr CharacterDataMonad.ptr_kinds_ptr_kinds_M h2
    using new_character_data_ptr_not_in_heap by blast
  then have "cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r"
    by simp
  then have "cast new_character_data_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r"
    by simp



  have object_ptr_kinds_eq_h:
    "object_ptr_kinds h2 = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
    using new_character_data_new_ptr h2 new_character_data_ptr by blast
  then have node_ptr_kinds_eq_h:
    "node_ptr_kinds h2 = node_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
    apply(simp add: node_ptr_kinds_def)
    by force
  then have character_data_ptr_kinds_eq_h:
    "character_data_ptr_kinds h2 = character_data_ptr_kinds h |∪| {|new_character_data_ptr|}"
    apply(simp add: character_data_ptr_kinds_def)
    by force
  have element_ptr_kinds_eq_h: "element_ptr_kinds h2 = element_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: node_ptr_kinds_def element_ptr_kinds_def)
  have document_ptr_kinds_eq_h: "document_ptr_kinds h2 = document_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: document_ptr_kinds_def)

  have object_ptr_kinds_eq_h2: "object_ptr_kinds h3 = object_ptr_kinds h2"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h",
                                OF set_val_writes h3])
    using set_val_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h2: "document_ptr_kinds h3 = document_ptr_kinds h2"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h2: "node_ptr_kinds h3 = node_ptr_kinds h2"
    using object_ptr_kinds_eq_h2
    by(auto simp add: node_ptr_kinds_def)

  have object_ptr_kinds_eq_h3: "object_ptr_kinds h' = object_ptr_kinds h3"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h",
                                OF set_disconnected_nodes_writes h'])
    using set_disconnected_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h3: "document_ptr_kinds h' = document_ptr_kinds h3"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h3: "node_ptr_kinds h' = node_ptr_kinds h3"
    using object_ptr_kinds_eq_h3
    by(auto simp add: node_ptr_kinds_def)



  have "document_ptr |∈| document_ptr_kinds h"
    using disc_nodes_h3 document_ptr_kinds_eq_h object_ptr_kinds_eq_h2
          get_disconnected_nodes_ptr_in_heap ‹type_wf h› document_ptr_kinds_def
    by (metis is_OK_returns_result_I)

  have children_eq_h: "⋀(ptr'::(_) object_ptr) children. ptr' ≠ cast new_character_data_ptr
                ⟹ h ⊢ get_child_nodes ptr' →_r children = h2 ⊢ get_child_nodes ptr' →_r children"
    using get_child_nodes_reads h2 get_child_nodes_new_character_data[rotated, OF new_character_data_ptr
h2]
    apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
```

```
    by blast+
  then have children_eq2_h:
    "⋀ptr'. ptr' ≠ cast new_character_data_ptr
      ⟹ |h ⊢ get_child_nodes ptr'|_r = |h2 ⊢ get_child_nodes ptr'|_r"
    using select_result_eq by force
  have object_ptr_kinds_eq_h:
    "object_ptr_kinds h2 = object_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
    using new_character_data_new_ptr h2 new_character_data_ptr by blast
  then have node_ptr_kinds_eq_h:
    "node_ptr_kinds h2 = node_ptr_kinds h |∪| {|cast new_character_data_ptr|}"
    apply(simp add: node_ptr_kinds_def)
    by force
  then have character_data_ptr_kinds_eq_h:
    "character_data_ptr_kinds h2 = character_data_ptr_kinds h |∪| {|new_character_data_ptr|}"
    apply(simp add: character_data_ptr_kinds_def)
    by force
  have element_ptr_kinds_eq_h: "element_ptr_kinds h2 = element_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: node_ptr_kinds_def element_ptr_kinds_def)
  have document_ptr_kinds_eq_h: "document_ptr_kinds h2 = document_ptr_kinds h"
    using object_ptr_kinds_eq_h
    by(auto simp add: document_ptr_kinds_def)

  have object_ptr_kinds_eq_h2: "object_ptr_kinds h3 = object_ptr_kinds h2"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h",
                                OF set_val_writes h3])
    using set_val_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h2: "document_ptr_kinds h3 = document_ptr_kinds h2"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h2: "node_ptr_kinds h3 = node_ptr_kinds h2"
    using object_ptr_kinds_eq_h2
    by(auto simp add: node_ptr_kinds_def)

  have object_ptr_kinds_eq_h3: "object_ptr_kinds h' = object_ptr_kinds h3"
    apply(rule writes_small_big[where P="λh h'. object_ptr_kinds h' = object_ptr_kinds h",
                                OF set_disconnected_nodes_writes h'])
    using set_disconnected_nodes_pointers_preserved
    by (auto simp add: reflp_def transp_def)
  then have document_ptr_kinds_eq_h3: "document_ptr_kinds h' = document_ptr_kinds h3"
    by (auto simp add: document_ptr_kinds_def)
  have node_ptr_kinds_eq_h3: "node_ptr_kinds h' = node_ptr_kinds h3"
    using object_ptr_kinds_eq_h3
    by(auto simp add: node_ptr_kinds_def)


  have "document_ptr |∈| document_ptr_kinds h"
    using disc_nodes_h3 document_ptr_kinds_eq_h object_ptr_kinds_eq_h2
          get_disconnected_nodes_ptr_in_heap ‹type_wf h› document_ptr_kinds_def
    by (metis is_OK_returns_result_I)

  have children_eq_h: "⋀(ptr'::(_) object_ptr) children. ptr' ≠ cast new_character_data_ptr
              ⟹ h ⊢ get_child_nodes ptr' →_r children = h2 ⊢ get_child_nodes ptr' →_r children"
    using get_child_nodes_reads h2 get_child_nodes_new_character_data[rotated, OF new_character_data_ptr
h2]
    apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
    by blast+
  then have children_eq2_h: "⋀ptr'. ptr' ≠ cast new_character_data_ptr
                              ⟹ |h ⊢ get_child_nodes ptr'|_r = |h2 ⊢ get_child_nodes ptr'|_r"
    using select_result_eq by force

  have "h2 ⊢ get_child_nodes (cast new_character_data_ptr) →_r []"
    using new_character_data_ptr h2 new_character_data_ptr_in_heap[OF h2 new_character_data_ptr]
```

```
        new_character_data_is_character_data_ptr[OF new_character_data_ptr]
        new_character_data_no_child_nodes
    by blast
  have disconnected_nodes_eq_h:
    "⋀doc_ptr disc_nodes. h ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                        = h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads h2
        get_disconnected_nodes_new_character_data[OF new_character_data_ptr h2]
    apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
    by blast+
  then have disconnected_nodes_eq2_h:
    "⋀doc_ptr. |h ⊢ get_disconnected_nodes doc_ptr|r = |h2 ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have children_eq_h2:
    "⋀ptr' children. h2 ⊢ get_child_nodes ptr' →r children = h3 ⊢ get_child_nodes ptr' →r children"
    using get_child_nodes_reads set_val_writes h3
    apply(rule reads_writes_preserved)
    by(auto simp add: set_val_get_child_nodes)
  then have children_eq2_h2:
    "⋀ptr'. |h2 ⊢ get_child_nodes ptr'|r = |h3 ⊢ get_child_nodes ptr'|r"
    using select_result_eq by force
  have disconnected_nodes_eq_h2:
    "⋀doc_ptr disc_nodes. h2 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                                         = h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads set_val_writes h3
    apply(rule reads_writes_preserved)
    by(auto simp add: set_val_get_disconnected_nodes)
  then have disconnected_nodes_eq2_h2:
    "⋀doc_ptr. |h2 ⊢ get_disconnected_nodes doc_ptr|r = |h3 ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have "type_wf h2"
    using ‹type_wf h› new_character_data_types_preserved h2 by blast
  then have "type_wf h3"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_val_writes h3]
    using  set_val_types_preserved
    by(auto simp add: reflp_def transp_def)
  then have "type_wf h'"
    using writes_small_big[where P="λh h'. type_wf h ⟶ type_wf h'", OF set_disconnected_nodes_writes
h']
    using  set_disconnected_nodes_types_preserved
    by(auto simp add: reflp_def transp_def)

  have children_eq_h3:
    "⋀ptr' children. h3 ⊢ get_child_nodes ptr' →r children = h' ⊢ get_child_nodes ptr' →r children"
    using get_child_nodes_reads set_disconnected_nodes_writes h'
    apply(rule reads_writes_preserved)
    by(auto simp add: set_disconnected_nodes_get_child_nodes)
  then have children_eq2_h3:
    " ⋀ptr'. |h3 ⊢ get_child_nodes ptr'|r = |h' ⊢ get_child_nodes ptr'|r"
    using select_result_eq by force
  have disconnected_nodes_eq_h3: "⋀doc_ptr disc_nodes. document_ptr ≠ doc_ptr
    ⟹ h3 ⊢ get_disconnected_nodes doc_ptr →r disc_nodes
                          = h' ⊢ get_disconnected_nodes doc_ptr →r disc_nodes"
    using get_disconnected_nodes_reads set_disconnected_nodes_writes h'
    apply(rule reads_writes_preserved)
    by(auto simp add: set_disconnected_nodes_get_disconnected_nodes_different_pointers)
  then have disconnected_nodes_eq2_h3: "⋀doc_ptr. document_ptr ≠ doc_ptr
            ⟹ |h3 ⊢ get_disconnected_nodes doc_ptr|r = |h' ⊢ get_disconnected_nodes doc_ptr|r"
    using select_result_eq by force

  have disc_nodes_document_ptr_h2: "h2 ⊢ get_disconnected_nodes document_ptr →r disc_nodes_h3"
```

```
  using disconnected_nodes_eq_h2 disc_nodes_h3 by auto
then have disc_nodes_document_ptr_h: "h ⊢ get_disconnected_nodes document_ptr →r disc_nodes_h3"
  using disconnected_nodes_eq_h by auto
then have "cast new_character_data_ptr ∉ set disc_nodes_h3"
  using ‹heap_is_wellformed h› using ‹cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|r›
        a_all_ptrs_in_heap_def heap_is_wellformed_def
  by (meson NodeMonad.ptr_kinds_ptr_kinds_M fset_mp fset_of_list_elem )

have "acyclic (parent_child_rel h)"
  using ‹heap_is_wellformed h›
  by (simp add: heap_is_wellformed_def acyclic_heap_def)
also have "parent_child_rel h = parent_child_rel h2"
proof(auto simp add: parent_child_rel_def)[1]
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|r"
  then show "a |∈| object_ptr_kinds h2"
    by (simp add: object_ptr_kinds_eq_h)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|r"
  then show "x ∈ set |h2 ⊢ get_child_nodes a|r"
    by (metis ObjectMonad.ptr_kinds_ptr_kinds_M
              ‹cast new_character_data_ptr ∉ set |h ⊢ object_ptr_kinds_M|r› children_eq2_h)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h2"
      and 1: "x ∈ set |h2 ⊢ get_child_nodes a|r"
  then show "a |∈| object_ptr_kinds h"
    using object_ptr_kinds_eq_h ‹h2 ⊢ get_child_nodes (cast new_character_data_ptr) →r []›
    by(auto)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h2"
    and 1: "x ∈ set |h2 ⊢ get_child_nodes a|r"
  then show "x ∈ set |h ⊢ get_child_nodes a|r"
    by (metis (no_types, lifting) ‹h2 ⊢ get_child_nodes (cast new_character_data_ptr) →r []›
              children_eq2_h empty_iff empty_set image_eqI select_result_I2)
qed
also have "... = parent_child_rel h3"
  by(auto simp add: parent_child_rel_def object_ptr_kinds_eq_h2 children_eq2_h2)
also have "... = parent_child_rel h'"
  by(auto simp add: parent_child_rel_def object_ptr_kinds_eq_h3 children_eq2_h3)
finally have "a_acyclic_heap h'"
  by (simp add: acyclic_heap_def)

have "a_all_ptrs_in_heap h"
  using ‹heap_is_wellformed h› by (simp add: heap_is_wellformed_def)
then have "a_all_ptrs_in_heap h2"
  apply(auto simp add: a_all_ptrs_in_heap_def)[1]
  using node_ptr_kinds_eq_h ‹cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|r›
        ‹h2 ⊢ get_child_nodes (cast new_character_data_ptr) →r []›
   apply (metis (no_types, hide_lams) children_eq_h fempty_iff fset_mp fset_of_list_simps(1)
               funionCI select_result_I2)
  by (simp add: disconnected_nodes_eq_h fset_rev_mp node_ptr_kinds_eq_h)
then have "a_all_ptrs_in_heap h3"
  by(auto simp add: a_all_ptrs_in_heap_def object_ptr_kinds_eq_h2 node_ptr_kinds_def
                    children_eq_h2 disconnected_nodes_eq_h2)
then have "a_all_ptrs_in_heap h'"
  apply(auto simp add: a_all_ptrs_in_heap_def object_ptr_kinds_eq_h3 node_ptr_kinds_def
                       children_eq_h3 )[1]
  using disconnected_nodes_eq_h3 object_ptr_kinds_eq_h object_ptr_kinds_eq_h2
```

```
    by (metis (no_types, lifting) disc_nodes_h3 finsertCI fset.map_comp fset_mp fset_of_list_elem
              funion_finsert_right h' local.set_disconnected_nodes_get_disconnected_nodes
              node_ptr_kinds_def node_ptr_kinds_eq_h select_result_I2 set_ConsD)

  have "⋀p. p |∈| object_ptr_kinds h ⟹ cast new_character_data_ptr ∉ set |h ⊢ get_child_nodes p|_r"
    using ‹heap_is_wellformed h› ‹cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r›
          heap_is_wellformed_children_in_heap
    by (meson NodeMonad.ptr_kinds_ptr_kinds_M a_all_ptrs_in_heap_def assms(3) assms(4) fset_mp
      fset_of_list_elem get_child_nodes_ok known_ptrs_known_ptr returns_result_select_result)
  then have "⋀p. p |∈| object_ptr_kinds h2 ⟹ cast new_character_data_ptr ∉ set |h2 ⊢ get_child_nodes
p|_r"
    using children_eq2_h
    apply(auto simp add: object_ptr_kinds_eq_h)[1]
    using ‹h2 ⊢ get_child_nodes (cast new_character_data_ptr) →_r []› apply auto[1]
    by (metis ObjectMonad.ptr_kinds_ptr_kinds_M ‹cast new_character_data_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r›)
  then have "⋀p. p |∈| object_ptr_kinds h3 ⟹ cast new_character_data_ptr ∉ set |h3 ⊢ get_child_nodes
p|_r"
    using object_ptr_kinds_eq_h2 children_eq2_h2 by auto
  then have new_character_data_ptr_not_in_any_children:
    "⋀p. p |∈| object_ptr_kinds h' ⟹ cast new_character_data_ptr ∉ set |h' ⊢ get_child_nodes p|_r"
    using object_ptr_kinds_eq_h3 children_eq2_h3 by auto

  have "a_distinct_lists h"
    using ‹heap_is_wellformed h›
    by (simp add: heap_is_wellformed_def)
  then have "a_distinct_lists h2"
    using ‹h2 ⊢ get_child_nodes (cast new_character_data_ptr) →_r []›
    apply(auto simp add: a_distinct_lists_def object_ptr_kinds_eq_h document_ptr_kinds_eq_h
        disconnected_nodes_eq2_h intro!: distinct_concat_map_I)[1]
      apply (metis distinct_sorted_list_of_set finite_fset sorted_list_of_set_insert)
     apply(case_tac "x=cast new_character_data_ptr")
      apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
     apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
    apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
    apply (metis IntI assms(1) assms(3) assms(4) empty_iff local.get_child_nodes_ok
      local.heap_is_wellformed_one_parent local.known_ptrs_known_ptr
      returns_result_select_result)
    apply(auto simp add: children_eq2_h[symmetric] insort_split dest: distinct_concat_map_E(2))[1]
    by (metis ‹local.a_distinct_lists h› ‹type_wf h2› disconnected_nodes_eq_h document_ptr_kinds_eq_h
        local.distinct_lists_no_parent local.get_disconnected_nodes_ok returns_result_select_result)
  then have "a_distinct_lists h3"
    by(auto simp add: a_distinct_lists_def disconnected_nodes_eq2_h2 document_ptr_kinds_eq_h2
                      children_eq2_h2 object_ptr_kinds_eq_h2)[1]
  then have "a_distinct_lists h'"
  proof(auto simp add: a_distinct_lists_def disconnected_nodes_eq2_h3 children_eq2_h3
                       object_ptr_kinds_eq_h3 document_ptr_kinds_eq_h3 intro!: distinct_concat_map_I)[1]
    fix x
    assume "distinct (concat (map (λdocument_ptr. |h3 ⊢ get_disconnected_nodes document_ptr|_r)
                      (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
      and "x |∈| document_ptr_kinds h3"
    then show "distinct |h' ⊢ get_disconnected_nodes x|_r"
      using document_ptr_kinds_eq_h3 disconnected_nodes_eq_h3 h' set_disconnected_nodes_get_disconnected_nodes
      by (metis (no_types, lifting) ‹cast new_character_data_ptr ∉ set disc_nodes_h3›
                ‹a_distinct_lists h3› ‹type_wf h'› disc_nodes_h3 distinct.simps(2)
                distinct_lists_disconnected_nodes get_disconnected_nodes_ok returns_result_eq
                returns_result_select_result)
  next
    fix x y xa
    assume "distinct (concat (map (λdocument_ptr. |h3 ⊢ get_disconnected_nodes document_ptr|_r)
                                        (sorted_list_of_set (fset (document_ptr_kinds h3)))))"
      and "x |∈| document_ptr_kinds h3"
      and "y |∈| document_ptr_kinds h3"
      and "x ≠ y"
```

```
    and "xa ∈ set |h' ⊢ get_disconnected_nodes x|_r"
    and "xa ∈ set |h' ⊢ get_disconnected_nodes y|_r"
  moreover have "set |h3 ⊢ get_disconnected_nodes x|_r ∩ set |h3 ⊢ get_disconnected_nodes y|_r = {}"
    using calculation by(auto dest: distinct_concat_map_E(1))
  ultimately show "False"
    apply(cases "x = document_ptr")
     apply (metis (no_types) NodeMonad.ptr_kinds_ptr_kinds_M ‹a_all_ptrs_in_heap h›
                             ‹cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r›
                             a_all_ptrs_in_heap_def assms(3) disc_nodes_h3
                             disconnected_nodes_eq2_h disconnected_nodes_eq2_h2
                             disconnected_nodes_eq2_h3 disjoint_iff_not_equal
                             document_ptr_kinds_eq_h document_ptr_kinds_eq_h2 fset_mp
                             fset_of_list_elem get_disconnected_nodes_ok h'
                             returns_result_select_result select_result_I2 set_ConsD
                             set_disconnected_nodes_get_disconnected_nodes)
    apply(cases "y = document_ptr" )
    apply (metis (no_types) NodeMonad.ptr_kinds_ptr_kinds_M
                ‹a_all_ptrs_in_heap h› ‹cast new_character_data_ptr ∉ set |h ⊢ node_ptr_kinds_M|_r›
                a_all_ptrs_in_heap_def assms(3) disc_nodes_h3 disconnected_nodes_eq2_h
                disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3 disjoint_iff_not_equal
                document_ptr_kinds_eq_h document_ptr_kinds_eq_h2 fset_mp fset_of_list_elem
                get_disconnected_nodes_ok h' returns_result_select_result select_result_I2 set_ConsD

                set_disconnected_nodes_get_disconnected_nodes)
    using disconnected_nodes_eq2_h3 by auto
next
  fix x xa xb
  assume 2: "(⋃x∈fset (object_ptr_kinds h3). set |h' ⊢ get_child_nodes x|_r)
             ∩ (⋃x∈fset (document_ptr_kinds h3). set |h3 ⊢ get_disconnected_nodes x|_r) = {}"
    and 3: "xa |∈| object_ptr_kinds h3"
    and 4: "x ∈ set |h' ⊢ get_child_nodes xa|_r"
    and 5: "xb |∈| document_ptr_kinds h3"
    and 6: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
  show "False"
    using disc_nodes_document_ptr_h disconnected_nodes_eq2_h3
    apply(cases "xb = document_ptr")
     apply (metis (no_types, hide_lams) "3" "4" "6"
                 ‹⋀p. p |∈| object_ptr_kinds h3 ⟹ cast new_character_data_ptr ∉ set |h3 ⊢ get_child_nodes
p|_r›
                 ‹a_distinct_lists h3› children_eq2_h3 disc_nodes_h3 distinct_lists_no_parent h'
                 select_result_I2 set_ConsD set_disconnected_nodes_get_disconnected_nodes)
    by (metis "3" "4" "5" "6" ‹a_distinct_lists h3› ‹type_wf h3› children_eq2_h3
              distinct_lists_no_parent get_disconnected_nodes_ok returns_result_select_result)
qed

have "a_owner_document_valid h"
  using ‹heap_is_wellformed h› by (simp add: heap_is_wellformed_def)
then have "a_owner_document_valid h'"
  using disc_nodes_h3 ‹document_ptr |∈| document_ptr_kinds h›
  apply(simp add: a_owner_document_valid_def)
  apply(simp add: object_ptr_kinds_eq_h object_ptr_kinds_eq_h3 )
  apply(simp add: object_ptr_kinds_eq_h2)
  apply(simp add: document_ptr_kinds_eq_h document_ptr_kinds_eq_h3 )
  apply(simp add: document_ptr_kinds_eq_h2)
  apply(simp add: node_ptr_kinds_eq_h node_ptr_kinds_eq_h3 )
  apply(simp add: node_ptr_kinds_eq_h2 node_ptr_kinds_eq_h )
  apply(auto simp add: children_eq2_h2[symmetric] children_eq2_h3[symmetric] disconnected_nodes_eq2_h

                       disconnected_nodes_eq2_h2 disconnected_nodes_eq2_h3)[1]
   apply (metis (no_types, lifting) document_ptr_kinds_eq_h h' list.set_intros(1)
                        local.set_disconnected_nodes_get_disconnected_nodes select_result_I2)
  apply(simp add: object_ptr_kinds_eq_h)
  by (metis (no_types, lifting) ObjectMonad.ptr_kinds_ptr_kinds_M
```

```
      ⟨cast_character_data_ptr2object_ptr new_character_data_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r⟩
      children_eq2_h disconnected_nodes_eq2_h3 document_ptr_kinds_eq_h h' list.set_intros(2)
      local.set_disconnected_nodes_get_disconnected_nodes select_result_I2)

  show "heap_is_wellformed h'"
    using ⟨a_acyclic_heap h'⟩ ⟨a_all_ptrs_in_heap h'⟩ ⟨a_distinct_lists h'⟩ ⟨a_owner_document_valid h'⟩
    by(simp add: heap_is_wellformed_def)
qed
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
  using instances
  by (auto simp add: l_create_character_data_wf_Core_DOM_def)
```

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 -
  obtain new_document_ptr where
    new_document_ptr: "h ⊢ new_document →_r new_document_ptr" and
    h': "h ⊢ new_document →_h h'"
    using assms(2)
    apply(simp add: create_document_def)
    using new_document_ok by blast
```

```
have "new_document_ptr ∉ set |h ⊢ document_ptr_kinds_M|_r"
  using new_document_ptr DocumentMonad.ptr_kinds_ptr_kinds_M
  using new_document_ptr_not_in_heap h' by blast
then have "cast new_document_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r"
  by simp

have "new_document_ptr |∉| document_ptr_kinds h"
  using new_document_ptr DocumentMonad.ptr_kinds_ptr_kinds_M
  using new_document_ptr_not_in_heap h' by blast
then have "cast new_document_ptr |∉| object_ptr_kinds h"
  by simp

have object_ptr_kinds_eq: "object_ptr_kinds h' = object_ptr_kinds h |∪| {|cast new_document_ptr|}"
  using new_document_new_ptr h' new_document_ptr by blast
then have node_ptr_kinds_eq: "node_ptr_kinds h' = node_ptr_kinds h"
  apply(simp add: node_ptr_kinds_def)
  by force
then have character_data_ptr_kinds_eq_h: "character_data_ptr_kinds h' = character_data_ptr_kinds h"
  by(simp add: character_data_ptr_kinds_def)
have element_ptr_kinds_eq_h: "element_ptr_kinds h' = element_ptr_kinds h"
  using object_ptr_kinds_eq
  by(auto simp add: node_ptr_kinds_def element_ptr_kinds_def)
have document_ptr_kinds_eq_h: "document_ptr_kinds h' = document_ptr_kinds h |∪| {|new_document_ptr|}"
  using object_ptr_kinds_eq
  apply(auto simp add: document_ptr_kinds_def)[1]
  by (metis (no_types, lifting) document_ptr_kinds_commutes document_ptr_kinds_def finsertI1 fset.map_comp)


have children_eq:
   "⋀(ptr'::(_) object_ptr) children. ptr' ≠ cast new_document_ptr
         ⟹ h ⊢ get_child_nodes ptr' →_r children = h' ⊢ get_child_nodes ptr' →_r children"
  using get_child_nodes_reads h' get_child_nodes_new_document[rotated, OF new_document_ptr h']
  apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
  by blast+
then have children_eq2: "⋀ptr'. ptr' ≠ cast new_document_ptr
                                ⟹ |h ⊢ get_child_nodes ptr'|_r = |h' ⊢ get_child_nodes ptr'|_r"
  using select_result_eq by force


have "h' ⊢ get_child_nodes (cast new_document_ptr) →_r []"
  using new_document_ptr h' new_document_ptr_in_heap[OF h' new_document_ptr]
              new_document_is_document_ptr[OF new_document_ptr] new_document_no_child_nodes
  by blast
have disconnected_nodes_eq_h:
  "⋀doc_ptr disc_nodes. doc_ptr ≠ new_document_ptr
  ⟹ h ⊢ get_disconnected_nodes doc_ptr →_r disc_nodes = h' ⊢ get_disconnected_nodes doc_ptr →_r disc_nodes"
  using get_disconnected_nodes_reads h' get_disconnected_nodes_new_document_different_pointers new_document_ptr
  apply(auto simp add: reads_def reflp_def transp_def preserved_def)[1]
  by (metis(full_types) ‹⋀thesis. (⋀new_document_ptr.
         ⟦h ⊢ new_document →_r new_document_ptr; h ⊢ new_document →_h h'⟧ ⟹ thesis) ⟹ thesis›
         local.get_disconnected_nodes_new_document_different_pointers new_document_ptr)+
then have disconnected_nodes_eq2_h: "⋀doc_ptr. doc_ptr ≠ new_document_ptr
              ⟹ |h ⊢ get_disconnected_nodes doc_ptr|_r = |h' ⊢ get_disconnected_nodes doc_ptr|_r"
  using select_result_eq by force
have "h' ⊢ get_disconnected_nodes new_document_ptr →_r []"
  using h' local.new_document_no_disconnected_nodes new_document_ptr by blast

have "type_wf h'"
  using ‹type_wf h› new_document_types_preserved h' by blast

have "acyclic (parent_child_rel h)"
  using ‹heap_is_wellformed h›
  by (simp add: heap_is_wellformed_def acyclic_heap_def)
```

```
also have "parent_child_rel h = parent_child_rel h'"
proof(auto simp add: parent_child_rel_def)[1]
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|_r"
  then show "a |∈| object_ptr_kinds h'"
    by (simp add: object_ptr_kinds_eq)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h"
    and 1: "x ∈ set |h ⊢ get_child_nodes a|_r"
  then show "x ∈ set |h' ⊢ get_child_nodes a|_r"
    by (metis ObjectMonad.ptr_kinds_ptr_kinds_M
            ‹cast new_document_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r› children_eq2)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h'"
     and 1: "x ∈ set |h' ⊢ get_child_nodes a|_r"
  then show "a |∈| object_ptr_kinds h"
    using object_ptr_kinds_eq ‹h' ⊢ get_child_nodes (cast new_document_ptr) →_r []›
    by(auto)
next
  fix a x
  assume 0: "a |∈| object_ptr_kinds h'"
    and 1: "x ∈ set |h' ⊢ get_child_nodes a|_r"
  then show "x ∈ set |h ⊢ get_child_nodes a|_r"
    by (metis (no_types, lifting) ‹h' ⊢ get_child_nodes (cast new_document_ptr) →_r []›
                                  children_eq2 empty_iff empty_set image_eqI select_result_I2)
qed
finally have "a_acyclic_heap h'"
  by (simp add: acyclic_heap_def)

have "a_all_ptrs_in_heap h"
  using ‹heap_is_wellformed h› by (simp add: heap_is_wellformed_def)
then have "a_all_ptrs_in_heap h'"
  apply(auto simp add: a_all_ptrs_in_heap_def)[1]
   apply (metis ObjectMonad.ptr_kinds_ptr_kinds_M
              ‹cast_document_ptr2object_ptr new_document_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r›
              ‹parent_child_rel h = parent_child_rel h'› assms(1) children_eq fset_of_list_elem
              local.heap_is_wellformed_children_in_heap local.parent_child_rel_child
              local.parent_child_rel_parent_in_heap node_ptr_kinds_eq)
  by (metis (no_types, lifting) ObjectMonad.ptr_kinds_ptr_kinds_M
              ‹cast_document_ptr2object_ptr new_document_ptr ∉ set |h ⊢ object_ptr_kinds_M|_r›
              ‹h' ⊢ get_child_nodes (cast_document_ptr2object_ptr new_document_ptr) →_r []›
              ‹parent_child_rel h = parent_child_rel h'› assms(1) disconnected_nodes_eq_h
              fset_of_list_elem h' local.heap_is_wellformed_disc_nodes_in_heap
              local.new_document_no_disconnected_nodes local.parent_child_rel_child
              local.parent_child_rel_parent_in_heap new_document_ptr node_ptr_kinds_eq
              select_result_I2)

have "a_distinct_lists h"
  using ‹heap_is_wellformed h›
  by (simp add: heap_is_wellformed_def)
then have "a_distinct_lists h'"
  using ‹h' ⊢ get_disconnected_nodes new_document_ptr →_r []›
        ‹h' ⊢ get_child_nodes (cast new_document_ptr) →_r []›

  apply(auto simp add: children_eq2[symmetric] a_distinct_lists_def insort_split object_ptr_kinds_eq
                       document_ptr_kinds_eq_h  disconnected_nodes_eq2_h intro!: distinct_concat_map_I)[1]
  apply (metis distinct_sorted_list_of_set finite_fset sorted_list_of_set_insert)

  apply(auto simp add:  dest: distinct_concat_map_E)[1]
  apply(auto simp add:  dest: distinct_concat_map_E)[1]
```

```
    using ‹new_document_ptr |∉| document_ptr_kinds h›
    apply(auto simp add: distinct_insort dest: distinct_concat_map_E)[1]
    using disconnected_nodes_eq_h
    apply (metis assms(1) assms(3) disconnected_nodes_eq2_h local.get_disconnected_nodes_ok
                         local.heap_is_wellformed_disconnected_nodes_distinct
                         returns_result_select_result)
  proof -
    fix x :: "(_) document_ptr" and y :: "(_) document_ptr" and xa :: "(_) node_ptr"
    assume a1: "x ≠ y"
    assume a2: "x |∈| document_ptr_kinds h"
    assume a3: "x ≠ new_document_ptr"
    assume a4: "y |∈| document_ptr_kinds h"
    assume a5: "y ≠ new_document_ptr"
    assume a6: "distinct (concat (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|_r)
                                            (sorted_list_of_set (fset (document_ptr_kinds h)))))"
    assume a7: "xa ∈ set |h' ⊢ get_disconnected_nodes x|_r"
    assume a8: "xa ∈ set |h' ⊢ get_disconnected_nodes y|_r"
    have f9: "xa ∈ set |h ⊢ get_disconnected_nodes x|_r"
      using a7 a3 disconnected_nodes_eq2_h by presburger
    have f10: "xa ∈ set |h ⊢ get_disconnected_nodes y|_r"
      using a8 a5 disconnected_nodes_eq2_h by presburger
    have f11: "y ∈ set (sorted_list_of_set (fset (document_ptr_kinds h)))"
      using a4 by simp
    have "x ∈ set (sorted_list_of_set (fset (document_ptr_kinds h)))"
      using a2 by simp
    then show False
      using f11 f10 f9 a6 a1 by (meson disjoint_iff_not_equal distinct_concat_map_E(1))
  next
    fix x xa xb
    assume 0: "h' ⊢ get_disconnected_nodes new_document_ptr →_r []"
      and 1: "h' ⊢ get_child_nodes (cast_document_ptr2object_ptr new_document_ptr) →_r []"
      and 2: "distinct (concat (map (λptr. |h ⊢ get_child_nodes ptr|_r)
                                          (sorted_list_of_set (fset (object_ptr_kinds h)))))"
      and 3: "distinct (concat (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|_r)
                                            (sorted_list_of_set (fset (document_ptr_kinds h)))))"
      and 4: "(⋃x∈fset (object_ptr_kinds h). set |h ⊢ get_child_nodes x|_r)
                    ∩ (⋃x∈fset (document_ptr_kinds h). set |h ⊢ get_disconnected_nodes x|_r) = {}"
      and 5: "x ∈ set |h ⊢ get_child_nodes xa|_r"
      and 6: "x ∈ set |h' ⊢ get_disconnected_nodes xb|_r"
      and 7: "xa |∈| object_ptr_kinds h"
      and 8: "xa ≠ cast_document_ptr2object_ptr new_document_ptr"
      and 9: "xb |∈| document_ptr_kinds h"
      and 10: "xb ≠ new_document_ptr"
    then show "False"

      by (metis ‹local.a_distinct_lists h› assms(3) disconnected_nodes_eq2_h
                local.distinct_lists_no_parent local.get_disconnected_nodes_ok
                 returns_result_select_result)
  qed

  have "a_owner_document_valid h"
    using ‹heap_is_wellformed h› by (simp add: heap_is_wellformed_def)
  then have "a_owner_document_valid h'"
    apply(auto simp add: a_owner_document_valid_def)[1]
    by (metis ‹cast_document_ptr2object_ptr new_document_ptr |∉| object_ptr_kinds h›
              ‹new_document_ptr |∉| document_ptr_kinds h› assms(3) assms(4) children_eq
              children_eq2 disconnected_nodes_eq2_h disconnected_nodes_eq_h
              is_OK_returns_result_E is_OK_returns_result_I local.get_child_nodes_ok
              local.get_child_nodes_ptr_in_heap local.get_disconnected_nodes_ok
              local.get_disconnected_nodes_ptr_in_heap local.known_ptrs_known_ptr node_ptr_kinds_eq)

  show "heap_is_wellformed h'"
    using ‹a_acyclic_heap h'› ‹a_all_ptrs_in_heap h'› ‹a_distinct_lists h'› ‹a_owner_document_valid h'›
```

```
    by(simp add: heap_is_wellformed_def)
qed
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
  using instances
  by (auto simp add: l_create_document_wf_Core_DOM_def)
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 create_heap

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

code_datatype ObjectClass.heap.Heap create_heap

lemma object_ptr_kinds_code1 [code]:
  "object_ptr_kinds (Heap (fmap_of_list xs)) = object_ptr_kinds (create_heap xs)"
  by(simp add: create_heap_def)

lemma object_ptr_kinds_code2 [code]:
  "object_ptr_kinds (create_heap xs) = fset_of_list (map fst xs)"
  by (simp add: object_ptr_kinds_def create_heap_def dom_map_of_conv_image_fst)

lemma object_ptr_kinds_code3 [code]:
  "fmlookup (the_heap (create_heap xs)) x = map_of xs x"
  by(auto simp add: create_heap_def fmlookup_of_list)

lemma object_ptr_kinds_code4 [code]:
  "the_heap (create_heap xs) = fmap_of_list xs"
  by(simp add: create_heap_def)

lemma object_ptr_kinds_code5 [code]:
  "the_heap (Heap x) = x"
  by simp

lemma object_ptr_kinds_code6 [code]:
  "noop = return ()"
  by(simp add: noop_def)
```

7.1.2 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_name =
      get_elements_by_tag_name ptr tag_name ⤜ 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_type)"
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 adoptNode (Document_adoptNode)

This theory contains the test cases for adoptNode.

```
theory Document_adoptNode
imports
  "Core_DOM_BaseTest"
begin

definition Document_adoptNode_heap :: "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit, unit)
heap" 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 :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where** *"document = Some (cast (document_ptr.Ref 1))"*

”Adopting an Element called ’x¡’ should work.”

lemma *"test (do {*

```
  tmp0 ← document . getElementsByTagName(''x<'');
  y ← return (tmp0 ! 0);
  child ← y . firstChild;
  tmp1 ← y . parentNode;
  tmp2 ← document . body;
  assert_equals(tmp1, tmp2);
  tmp3 ← y . ownerDocument;
  assert_equals(tmp3, document);
  tmp4 ← 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);
  tmp8 ← child . ownerDocument;
  assert_equals(tmp8, 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"
```

by *eval*

”Adopting an Element called ’:good:times:’ should work.”

lemma *"test (do {*

```
  x ← document . createElement('':good:times:'');
  tmp0 ← 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"
  by eval
```

end

7.3 Testing getElementById (Document_getElementById)

This theory contains the test cases for getElementById.

```
theory Document_getElementById
imports
  "Core_DOM_BaseTest"
begin

definition Document_getElementById_heap :: "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit,
unit) heap" 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 :: "(unit, unit, unit, unit, unit, unit) object_ptr option" where "document = Some (cast
(document_ptr.Ref 1))"
```

”Document.getElementById with a script-inserted element”

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test2'';
  test ← document . createElement(''div'');
  test . setAttribute(''id'', TEST_ID);
  gBody . appendChild(test);
  result ← 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(TEST_ID);
  assert_equals(removed, None, ''should not get removed element.'')
}) Document_getElementById_heap"
  by eval
```

”update ‘id‘ attribute via setAttribute/removeAttribute”

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test3'';
  test ← document . createElement(''div'');
  test . setAttribute(''id'', TEST_ID);
  gBody . appendChild(test);
  UPDATED_ID ← return ''test3-updated'';
  test . setAttribute(''id'', UPDATED_ID);
  e ← document . getElementById(UPDATED_ID);
  assert_equals(e, test, ''should get the element with id.'');
  old ← document . getElementById(TEST_ID);
  assert_equals(old, None, ''shouldn't get the element by the old id.'');
  test . removeAttribute(''id'');
  e2 ← document . getElementById(UPDATED_ID);
  assert_equals(e2, None, ''should return null when the passed id is none in document.'')
}) Document_getElementById_heap"
  by eval
```

”Ensure that the id attribute only affects elements present in a document”

```
lemma "test (do {
  TEST_ID ← return ''test4-should-not-exist'';
  e ← document . createElement(''div'');
  e . setAttribute(''id'', TEST_ID);
  tmp0 ← document . getElementById(TEST_ID);
  assert_equals(tmp0, None, ''should be null'');
  tmp1 ← document . body;
  tmp1 . appendChild(e);
  tmp2 ← document . getElementById(TEST_ID);
  assert_equals(tmp2, e, ''should be the appended element'')
}) Document_getElementById_heap"
  by eval
```

”in tree order, within the context object's tree”

```
lemma "test (do {
  gBody ← document . body;
```

```
  TEST_ID ← return ''test5'';
  target ← 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 . createElement(''div'');
  element4 . setAttribute(''id'', TEST_ID);
  element4 . setAttribute(''data-name'', ''4th'');
  gBody . appendChild(element4);
  target2 ← 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(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"
  by eval
```

"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 . createElement(''div'');
  s . setAttribute(''id'', TEST_ID);
  tmp0 ← document . createElement(''div'');
  tmp0 . appendChild(s);
  tmp1 ← document . getElementById(TEST_ID);
  assert_equals(tmp1, None, ''should be null'')
}) Document_getElementById_heap"
  by eval
```

"changing attribute's value via 'Attr' gotten from 'Element.attribute'."

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test7'';
  element ← document . createElement(''div'');
  element . setAttribute(''id'', TEST_ID);
  gBody . appendChild(element);
  target ← 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(TEST_ID);
  assert_equals(target2, None, ''should return null after updated id via Attr.value'');
  target3 ← document . getElementById((TEST_ID @ ''-updated''));
  assert_equals(target3, element, ''should be equal to the updated element.'')
}) Document_getElementById_heap"
  by eval
```

"update 'id' attribute via element.id"

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test12'';
  test ← 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(UPDATED_ID);
  assert_equals(e, test, ''should get the element with id.'');
```

```
  old ← document . getElementById(TEST_ID);
  assert_equals(old, None, ''shouldn't get the element by the old id.'');
  test . setAttribute(''id'', '''');
  e2 ← document . getElementById(UPDATED_ID);
  assert_equals(e2, None, ''should return null when the passed id is none in document.'')
}) Document_getElementById_heap"
  by eval
```

"where insertion order and tree order don't match"

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test13'';
  container ← document . createElement(''div'');
  container . setAttribute(''id'', (TEST_ID @ ''-fixture''));
  gBody . appendChild(container);
  element1 ← document . createElement(''div'');
  element1 . setAttribute(''id'', TEST_ID);
  element2 ← document . createElement(''div'');
  element2 . setAttribute(''id'', TEST_ID);
  element3 ← document . createElement(''div'');
  element3 . setAttribute(''id'', TEST_ID);
  element4 ← 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(TEST_ID);
  assert_equals(test, element1, ''should return 1st element'');
  container . removeChild(element1);
  test ← document . getElementById(TEST_ID);
  assert_equals(test, element2, ''should return 2nd element'');
  container . removeChild(element2);
  test ← document . getElementById(TEST_ID);
  assert_equals(test, element3, ''should return 3rd element'');
  container . removeChild(element3);
  test ← document . getElementById(TEST_ID);
  assert_equals(test, element4, ''should return 4th element'');
  container . removeChild(element4)
}) Document_getElementById_heap"
  by eval
```

"Inserting an id by inserting its parent node"

```
lemma "test (do {
  gBody ← document . body;
  TEST_ID ← return ''test14'';
  a ← document . createElement(''a'');
  b ← document . createElement(''b'');
  a . appendChild(b);
  b . setAttribute(''id'', TEST_ID);
  tmp0 ← document . getElementById(TEST_ID);
  assert_equals(tmp0, None);
  gBody . appendChild(a);
  tmp1 ← document . getElementById(TEST_ID);
  assert_equals(tmp1, b)
}) Document_getElementById_heap"
  by eval
```

"Document.getElementById must not return nodes not present in document"

```
lemma "test (do {
  TEST_ID ← return ''test15'';
  outer ← document . getElementById(''outer'');
  middle ← document . getElementById(''middle'');
```

```
  inner ← document . getElementById(''inner'');
  tmp0 ← document . getElementById(''middle'');
  outer . removeChild(tmp0);
  new_el ← document . createElement(''h1'');
  new_el . setAttribute(''id'', ''heading'');
  inner . appendChild(new_el);
  tmp1 ← document . getElementById(''heading'');
  assert_equals(tmp1, None)
}) Document_getElementById_heap"
  by eval
```

end

7.4 Testing insertBefore (Node_insertBefore)

This theory contains the test cases for insertBefore.

theory *Node_insertBefore*
imports
"Core_DOM_BaseTest"
begin

definition *Node_insertBefore_heap :: "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit, unit) heap"* **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 *document :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where** *"document = Some (cast (document_ptr.Ref 1))"*

”Calling insertBefore an a leaf node Text must throw HIERARCHY_REQUEST_ERR.”

lemma *"test (do {*
node ← document . createTextNode(''Foo'');
tmp0 ← document . createTextNode(''fail'');
assert_throws(HierarchyRequestError, node . insertBefore(tmp0, None))
}) Node_insertBefore_heap"
by eval

”Calling insertBefore with an inclusive ancestor of the context object must throw HIERARCHY_REQUEST_ERR.”

lemma *"test (do {*
tmp1 ← document . body;
tmp2 ← document . getElementById(''log'');
tmp0 ← document . body;
assert_throws(HierarchyRequestError, tmp0 . insertBefore(tmp1, tmp2));

```
  tmp4 ← document . documentElement;
  tmp5 ← document . getElementById(''log'');
  tmp3 ← document . body;
  assert_throws(HierarchyRequestError, tmp3 . insertBefore(tmp4, tmp5))
}) Node_insertBefore_heap"
  by eval
```

"Calling insertBefore with a reference child whose parent is not the context node must throw a NotFoundError."

```
lemma "test (do {
  a ← document . createElement(''div'');
  b ← document . createElement(''div'');
  c ← document . createElement(''div'');
  assert_throws(NotFoundError, a . insertBefore(b, c))
}) Node_insertBefore_heap"
  by eval
```

"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"
  by eval
```

"Inserting a node before itself should not move the node"

```
lemma "test (do {
  a ← document . createElement(''div'');
  b ← document . createElement(''div'');
  c ← 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"
  by eval
```

end

7.5 Testing removeChild (Node_removeChild)

This theory contains the test cases for removeChild.

```
theory Node_removeChild
imports
  "Core_DOM_BaseTest"
begin

definition Node_removeChild_heap :: "(unit, unit, unit, unit, unit, unit, unit, unit, unit, unit, unit) heap"
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 *document :: "(unit, unit, unit, unit, unit, unit) object_ptr option"* **where** *"document = Some (cast (document_ptr.Ref 1))"*

"Passing a detached Element to removeChild should not affect it."

lemma *"test (do {*
  doc ← return document;
  s ← doc . createElement(''div'');
  tmp0 ← s . ownerDocument;
  assert_equals(tmp0, doc);
  tmp1 ← document . body;
  assert_throws(NotFoundError, tmp1 . removeChild(s));
  tmp2 ← s . ownerDocument;
  assert_equals(tmp2, doc)
}) Node_removeChild_heap"
  by *eval*

"Passing a non-detached Element to removeChild should not affect it."

lemma *"test (do {*
  doc ← return document;
  s ← doc . createElement(''div'');
  tmp0 ← doc . documentElement;
  tmp0 . appendChild(s);
  tmp1 ← s . ownerDocument;
  assert_equals(tmp1, doc);
  tmp2 ← document . body;
  assert_throws(NotFoundError, tmp2 . removeChild(s));
  tmp3 ← s . ownerDocument;
  assert_equals(tmp3, doc)
}) Node_removeChild_heap"
  by *eval*

"Calling removeChild on an Element with no children should throw NOT_FOUND_ERR."

lemma *"test (do {*
  doc ← return 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"
  by eval
```

"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 ← document . body;
  assert_throws(NotFoundError, tmp1 . removeChild(s));
  tmp2 ← s . ownerDocument;
  assert_equals(tmp2, doc)
}) Node_removeChild_heap"
  by eval
```

"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 ← document . body;
  assert_throws(NotFoundError, tmp2 . removeChild(s));
  tmp3 ← s . ownerDocument;
  assert_equals(tmp3, doc)
}) Node_removeChild_heap"
  by eval
```

"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"
  by eval
```

"Passing a value that is not a Node reference to removeChild should throw TypeError."

```
lemma "test (do {
  tmp0 ← document . body;
  assert_throws(TypeError, tmp0 . removeChild(None))
}) Node_removeChild_heap"
  by eval
```

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.