Reasoning about Object Capabilities with Logical Relations and Effect Parametricity
Technical Report including Proofs and Details

Dominique Devriese
Lars Birkedal
Frank Piessens

Report CW 690, January 2016

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Reasoning about Object Capabilities with Logical Relations and Effect Parametricity
Technical Report including Proofs and Details

Dominique Devriese
Lars Birkedal
Frank Piessens

Report CW 690, January 2016

Department of Computer Science, KU Leuven

Abstract

Object capabilities are a technique for fine-grained privilege separation in programming languages and systems, with important applications in security. However, current formal characterisations do not fully capture capability-safety of a programming language and are not sufficient for verifying typical applications. Using state-of-the-art techniques from programming languages research, we define a logical relation for a core calculus of JavaScript that better characterises capability-safety. The relation is powerful enough to reason about typical capability patterns and supports evolvable invariants on shared data structures, capabilities with restricted authority over them and isolated components with restricted communication channels. We use a novel notion of effect parametricity for deriving properties about effects. Our results imply memory access bounds that have previously been used to characterise capability-safety. This is a technical report accompanying a paper by the same title and authors, which contains an additional section about a binary version of our results, as well as proofs and details for our results.

Keywords : object capabilities, effect parametricity, capability-safety, logical relations.

Reasoning about Object Capabilities with Logical Relations and Effect Parametricity Technical Report including Proofs and Details

Dominique Devriese
iMinds-DistriNet, KU Leuven, Belgium
Email: dominique.devriese@cs.kuleuven.be

Lars Birkedal
Aarhus University, Denmark
Email: birkedal@cs.au.dk

Frank Piessens
iMinds-DistriNet, KU Leuven, Belgium
Email: frank.piessens@cs.kuleuven.be

Abstract—**Object capabilities are a technique for fine-grained privilege separation in programming languages and systems, with important applications in security. However, current formal characterisations do not fully capture capability-safety of a programming language and are not sufficient for verifying typical applications. Using state-of-the-art techniques from programming languages research, we define a logical relation for a core calculus of JavaScript that better characterises capability-safety. The relation is powerful enough to reason about typical capability patterns and supports evolvable invariants on shared data structures, capabilities with restricted authority over them and isolated components with restricted communication channels. We use a novel notion of effect parametricity for deriving properties about effects. Our results imply memory access bounds that have previously been used to characterise capability-safety. This is a technical report accompanying a paper by the same title and authors [1], which contains an additional section about a binary version of our results, as well as proofs and details for our results.**

1. Introduction

Privilege separation between components and the Principle Of Least Authority (POLA) are key ingredients for constructing secure systems. Since decades, systems with *object capabilities* [2], [3] have supported a very fine-grained separation of privileges. Lately, a renewed research interest in the technique has produced implementations at the level of the OS [4], the processor [5] and the programming language [3], [6]–[9]. Applications of the technique include sandboxing untrusted code [7], [10], efficient security auditing [6], fault isolation [4]. The technique also has potential applications outside security (e.g. enabling testing in simulated environments, enforcing architectural choices), but these have not drawn as much attention so far.

In the object capabilities model, effects can only be produced by sending messages to *objects*. These can be objects in the sense of object-oriented programming, but not necessarily so. *Device objects* are primitive objects that model resources in the outside world and produce effects in the outside world when receiving a message. *Instance objects* are programmer-defined objects that may hold private state (data or references to other objects). They execute programmer-defined code upon receipt of a message and this may include sending messages to other objects. References to objects are called *capabilities*, as they represent the authority to invoke methods on those objects.

A *capability-safe* programming language implements certain restrictions to enforce the object-capability model. Such a language guarantees that sending messages to objects is indeed the *only* way to produce effects and that capabilities cannot be forged. Additionally, the language excludes globally accessible mutable state to provide control over the capabilities that a component starts out with. These restrictions enable low-cost fine-grained privilege separation; a programmer defines the authority of a component by controlling the capabilities it holds.

To make this more concrete, consider a web page embedding an untrusted advertisement. The advertisement's initialisation requires access to the DOM for at least the ad's designated location on the page. However, it should be prevented from accessing or modifying other parts of the DOM. In a capability-safe JavaScript-like language, the web page could use the following function rnode to construct a restricted capability for accessing a subtree of the DOM:

$$
\begin{array}{l}
\mathit{rnode} \overset{\mathsf{def}}{=} \mathtt{func}(\mathit{node}, d) \\
\qquad \left\{ \begin{array}{r}
\mathit{getChild} = \mathtt{func}(\mathit{id}) \left\{ \mathit{rnode}(\mathit{node}.\mathit{getChild}(\mathit{id}), d+1) \right\} \\
\mathit{parent} = \mathtt{func}() \left\{ \begin{array}{r} \mathtt{if}\ (d \leq 0)\ \{\mathtt{error}\}\ \mathtt{else} \\ \{\mathit{rnode}(\mathit{node}.\mathit{parent}(), d-1)\} \end{array} \right\} \\
\mathit{getProp} = \mathtt{func}(\mathit{id})\{\mathit{node}.\mathit{getProp}(\mathit{id})\} \\
\mathit{setProp} = \mathtt{func}(\mathit{id}, v)\{\mathit{node}.\mathit{setProp}(\mathit{id}, v)\} \\
\mathit{addChild} = \mathtt{func}(\mathit{id}) \left\{ \mathit{rnode}(\mathit{node}.\mathit{addChild}(\mathit{id}), d+1) \right\} \\
\mathit{delChild} = \mathtt{func}(\mathit{id})\{\mathit{node}.\mathit{delChild}(\mathit{id})\}
\end{array} \right\}
\end{array}
$$

The function wraps a DOM node and forwards invocations to hypothetical $\mathit{getChild}$, $\mathit{parent}\ldots$ methods, but prevents access to nodes more than d levels higher in the tree.

Now assume the next web page initialisation function:

$$\mathit{initWebPage} \overset{\mathsf{def}}{=} \mathtt{func}(\mathit{document}, \mathit{ad})$$

$$\left\{\begin{array}{l} \mathit{document.setProp}(\text{“}\mathit{someProperty}\text{”}, 42) \\ \texttt{let}\ (\mathit{adNode} = \mathit{document.addChild}(\text{“}\mathit{ad\_div}\text{”})) \\ \texttt{let}\ (\mathit{rAdNode} = \mathit{rnode}(\mathit{adNode}, 0)) \\ \mathit{ad.initialize}(\mathit{rAdNode}) \\ \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42 \end{array}\right\}$$

A capability-safe language enforces the encapsulation of the $\mathit{rAdNode}$ object, prevents direct access to $\mathit{document}$ through other channels and rules out mutable global state which could (have been made to) contain a reference to $\mathit{document}$. If we assume sane behaviour of $\mathit{document}$ and that ad has no capabilities besides $\mathit{rAdNode}$ to begin with, then we can convince ourselves that if $\mathit{initWebPage}$ terminates, it must return `true`. But can we make this reasoning precise?

Formal reasoning about code in object capability languages requires a good understanding of the model. What exactly does it mean that a language is *capability-safe*? What guarantees can we provide about code written in it and how can we formally prove properties for maximum assurance? Several researchers have worked on this [3], [10], [11], but to this day, our understanding of the model is not satisfactory.

The problem is that previous research (except Spiessens [11], [12], but see Section 7) focuses on *reference graph* dynamics. For a state in a program's execution, the reference graph is the graph with the allocated objects as nodes, and the references they hold to each other as edges. Properties like *No Authority Amplification* and *Only Connectivity Begets Connectivity* [3], [10] restrict how the reference graph can evolve, depending on the references held by the executing code.

Unfortunately, these properties offer only a conservative bound on components' authority: the *topology-only bound*. It is based on syntactic structure of objects (whether or not they contain a reference to each other) and ignores their *behaviour*. This limitation is important, because a key quality of the model is the ability to define *custom capabilities* as instance objects. Such capabilities (like our $\mathit{rAdNode}$) restrict other capabilities, make them conditional or revocable or otherwise modify and combine them. The object capabilities community has studied many patterns for defining custom capabilities, but their soundness typically does not follow from topology-only bounds on memory access.

Contributions and Outline In this paper, we propose a novel, more semantic approach for reasoning about object capability languages. The key contribution is a step-indexed Kripke logical relation [13], [14] with two key features: support for reasoning about custom effect properties using *effect parametricity* (EP) and support for reasoning about shared data structures with evolvable invariants and authority using a novel form of possible worlds. Because we use many concepts that may be new to a security audience, we provide a gradual introduction in Section 2, where we introduce (first) logical relations and effect parametricity and (second) step-indexed logical relations for two simple languages without a shared mutable store. Section 3 repeats the definition of λ_{JS}, a pre-existing core calculus for JavaScript [15] and presents the logical relation capturing its capability-safety. It uses ideas and techniques from previous work [16]–[18] but also novel ideas, in particular a variation on Dreyer et al.'s public/private transitions [17] for modelling authority over shared data and our notion of *effect parametricity* to support custom properties about effects. We demonstrate in Section 4 that the logical relation can be used to verify non-trivial examples involving evolvable invariants on shared data structures, restricted authority over them (like the web-page-with-an-ad example above) and isolating components with restricted communication channels. These examples show that, contrary to previous work on capability-safety, our results capture more than just the topology-only bound on authority. To demonstrate that our results encompass previous results, Section 6 derives such standard syntactical bounds after first repeating a formulation of the properties by Maffeis et al. in Section 5. Section 7 discusses related work and Section 8 concludes. In Appendix A of this Technical Report, we provide more details about the relational generalisation of our results. Appendices B, C, D, E, F, G and H contain technical details and proofs of the results in the paper.

2. A Simpler Setting

In order to explain our work to a wide audience, this section gradually introduces concepts and techniques for two very simple languages, as preparation for the subset of λ_{JS} that we work with later on. The first is $\lambda_{\mathtt{out},FO}$, a simple first-order calculus (no lambdas) which we then extend to the higher-order $\lambda_{\mathtt{out},HO}$. Both contain only a single primitive capability, for producing textual output.

Contrary to λ_{JS}, neither calculus has a mutable store. This reduces the technicalities for reasoning, but makes the results less interesting. Without shared mutable data, there is no point in defining invariants on it or authority over it and less ways to pass capabilities around. What remains is *effect parametricity*: a general property that allows proving custom properties about untrusted expressions' effects.

Although $\lambda_{\mathtt{out},FO}$ allows introducing our techniques in a very simple setting, it lacks the ability to define custom capabilities, limiting the value of effect parametricity. This limitation is removed in $\lambda_{\mathtt{out},HO}$, where we define and prove correct a simple custom capability that restricts output to upper-case. This simple custom capability models practical scenarios where untrusted code in the browser might receive a capability for injecting only valid HTML in the web page or evaluating a safe subset of JavaScript.

The definitions in this section are an instance of a general technique for reasoning about higher-order programming languages known as *logical relations* (LRs; see e.g. [13], [14], [16], [17]). Our LRs are *unary*, because we use predicates rather than relations. We do not assume prior knowledge about LRs and motivate and introduce the techniques gradually throughout the text.

Syntax:

$$\begin{array}{rcl} e \in \mathit{Expr} & ::= & x \mid v \mid \mathtt{let}(x = e)e \mid \mathtt{if}(e)\{e\}\ \mathtt{else}\ \{e\} \\ & \mid & e; e \mid \mathtt{while}(e)\{e\} \mid e.\mathtt{print}(e) \\ v \in \mathit{Val} & ::= & \mathit{num} \mid \mathit{str} \mid \mathit{bool} \mid \mathtt{undef} \mid \mathtt{null} \mid \mathtt{out} \\ E & ::= & \cdot \mid \mathtt{let}(x = E)\ e \mid \mathtt{if}(E)\{e\}\ \mathtt{else}\ \{e\} \\ & \mid & E; e \mid E.\mathtt{print}(e) \mid v.\mathtt{print}(E) \end{array}$$

Pure evaluations:

$$\mathtt{if}(\mathtt{true})\{e_1\}\ \mathtt{else}\ \{e_2\} \hookrightarrow e_1 \quad \text{(E-IfTrue)}$$

$$\mathtt{if}\ (\mathtt{false})\{e_1\}\ \mathtt{else}\ \{e_2\} \hookrightarrow e_2 \quad \text{(E-IfFalse)}$$

$$v; e \hookrightarrow e \quad \text{(E-Begin-Discard)} \qquad \mathtt{let}\ (x = v)\ e \hookrightarrow e[x/v] \quad \text{(E-Let)}$$

$$\begin{array}{l} \mathtt{while}\ (e_1)\{e_2\} \hookrightarrow \\ \quad \mathtt{if}\ (e_1)\{e_2; \mathtt{while}(e_1)\{e_2\}\}\ \mathtt{else}\ \{\mathtt{undef}\} \end{array} \quad \text{(E-While)}$$

$$\frac{e_1 \hookrightarrow e_2}{E\langle e_1\rangle \to E\langle e_2\rangle} \quad \text{(E-Cxt)}$$

Impure evaluations:

$$\frac{e_1 \to e_2}{e_1 \to_{[]} e_2} \quad \text{(E-Pure)}$$

$$E\langle \mathtt{out}.\mathtt{print}(\mathit{str})\rangle \to_{[\mathit{str}]} E\langle \mathtt{undef}\rangle \quad \text{(E-Out)}$$

Figure 1. $\lambda_{\mathtt{out},FO}$, a first-order calculus with an output capability.

2.1. An output capability in a first-order calculus

We define $\lambda_{\mathtt{out},FO}$ in Figure 1. It is a simple imperative untyped programming language with expressions e and values v. The (small-step) operational semantics is split in two parts: pure evaluations $e \to e'$ and impure evaluations $e \to_o e'$. The latter may produce textual output in the list of strings o. Multi-step impure evaluations $e \to_o^* e'$, concatenate the outputs of the individual steps. The pure evaluation is defined in terms of primitive pure evaluations $e \hookrightarrow e'$ and evaluation contexts E that produce a strict evaluation order. There are standard `if`, `while`, `let` and sequence ($e; e'$) expressions. Output is produced by invoking a `print` method on the primitive capability `out`. Note that `out` is an internal value; similar to an object reference (`0x1234`) in a language like Java, programmers cannot write it but the language runtime may give access to it during execution, for example as an argument to a program's main. Note also that $\lambda_{\mathtt{out},FO}$ provides plenty of opportunity for stuck terms, e.g. an `if` branching on a non-boolean or invoking `print` on `true`. We could also throw exceptions in such cases, but stuck expressions are simpler.

Let us now formulate our Fundamental Theorem, guaranteeing effect parametricity. Intuitively, the theorem states that if an *acceptability* property (e.g. only producing uppercase output) holds for the possible effects of the values that an expression has access to (e.g. the mentioned custom capability), then it must also hold for the expression as a whole. These acceptability properties can be chosen freely, as long as they are *admissible*. To understand the conditions for admissibility, one should understand that not every property is enforcable using the general approach of taking arbitrary untrusted code and controlling the capabilities that it has access to. For example, no matter what capabilities a piece of code has access to, it cannot be forced to return a specific value, so admissible effect properties should not distinguish expressions by their return values.

To formalise this, we need to put in place some technical machinery: we (a) define the properties on expressions that we work with, (b) define when such a property is "admissible" and (c) formulate the actual theorem, in terms of an expression's scope. Let us consider each of the steps.

Semantic properties First, we limit the properties we work with to *semantic* properties, which only consider the meaning of an expression and not syntactic artefacts. We do this by treating as equivalent all expressions that purely evaluate to the same expression and we only consider properties on *commands*: a syntactic class of expressions that are the intended end result of *pure* evaluations (like *values* are the intended end result of impure evaluations). In addition to values, this also includes expressions like $\mathtt{out}.\mathtt{print}(\text{“}\mathit{abc}\text{”})$ which are normal w.r.t. pure but not w.r.t. impure evaluation. Formally, we define a command as either a value or an expression $E\langle \mathit{cmd}_0\rangle$, blocked on a `print` call cmd_0:

$$\mathit{cmd}_0 ::= v.\mathtt{print}(v) \qquad \mathit{cmd} \in \mathit{Cmd} ::= E\langle \mathit{cmd}_0\rangle \mid v$$

We will work with *command predicates*, i.e. subsets of commands $P \in \mathcal{P}(\mathit{Cmd})$. The *expression extension* $\mathcal{E}[P]$ of such a P accepts an expression e if commands that it evaluates to are in P.

$$\mathcal{E}[P] \overset{\mathsf{def}}{=} \{e \mid \text{If } e \to^* \mathit{cmd}, \text{ then } \mathit{cmd} \in P\}$$

A first command predicate is $\mathit{PureVal}$, containing only pure values, i.e. numbers, strings, booleans, `undef` or `null`:

$$\mathit{PureVal} \overset{\mathsf{def}}{=} \mathit{Num} \cup \mathit{Str} \cup \mathit{Bool} \cup \{\mathtt{undef}, \mathtt{null}\}.$$

Properties about effects As explained above, effect parametricity is concerned with *admissible* effect properties that are in principle enforceable using the object capability approach. We formalise this in a very general way by taking inspiration from *monads* (used to model effectful computations in pure functional languages like Haskell [19], [20]). We define an *effect interpretation* as a couple (μ, ρ), where $\mu \in \mathcal{P}(\mathit{Val}) \to \mathcal{P}(\mathit{Cmd})$ and $\rho \in \mathcal{P}(\mathit{Cap})$ with $\mathit{Cap} = \{\mathtt{out}\}$ the set of primitive capabilities. Intuitively, for a given value predicate P of acceptable result values, the command predicate $\mu\ P$ defines a set of expressions that impurely evaluate to values in P. Typically, $\mu\ P$ will not contain all such expressions, but only those that produce effects deemed (in some way) acceptable. ρ defines a set of acceptable primitive capabilities. Defined in terms of the effect interpretation (μ, ρ), the value predicate $\mathit{Val}_{\mu,\rho}$ defines the full set of acceptable λ_{JS} values: pure values and primitive capabilities in ρ. $\mathit{Val}_{\mu,\rho} \overset{\mathsf{def}}{=} \mathit{PureVal} \cup \rho$.[1]

The couple (μ, ρ) is *admissible* or *valid* if it satisfies three conditions or *axioms*. We list them and explain below:[2]

1. Note that $\mathit{Val}_{\mu,\rho}$ does not depend on μ, but that will change later.

2. Readers familiar with monads may recognise a correspondence between A-Pure and A-Bind and the monad operations *return* and *bind*.

$\Gamma ::= \emptyset \mid \Gamma, x$

$$\frac{v \neq \texttt{out}}{\Gamma \vdash v} \qquad \frac{\Gamma \vdash e_1 \quad \Gamma, x \vdash e_2}{\Gamma \vdash \texttt{let}(x = e_1)\ e_2} \qquad \frac{x \in \Gamma}{\Gamma \vdash x}$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2 \quad \Gamma \vdash e_3}{\Gamma \vdash \texttt{if}(e_1)\{e_2\}\,\texttt{else}\,\{e_3\}} \qquad \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1; e_2}$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash \texttt{while}(e_1)\{e_2\}} \qquad \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1.\texttt{print}(e_2)}$$

Figure 2. Well-scopedness of expressions in $\lambda_{\texttt{out},FO}$.

- A-PURE: For a value $v \in P$, v must also be in $\mu\ P$.
- A-BIND: If $cmd \in \mu\ P$ and $E\langle v\rangle \in \mathcal{E}[\mu\ P']$ for all values $v \in P$, then $E\langle cmd\rangle \in \mathcal{E}[\mu\ P']$.
- A-PRINT: If $v \in Val_{\mu,\rho}$ then $v.\texttt{print}(v') \in \mu\ Val_{\mu,\rho}$.

Axiom A-PURE states that it must be acceptable to produce no effects: a value in P must also be in $\mu\ P$. For the second axiom, we consider a command cmd in $\mu\ P$ (i.e. producing acceptable effects and a result in P) and an execution context E such that $E\langle v\rangle$ is in $\mu\ P'$ (i.e. produces acceptable effects and a result in P') for any v in P (i.e. any possible result of cmd). Then Axiom A-BIND requires that the composed expression $E\langle cmd\rangle$ should be in $\mathcal{E}[\mu\ P']$. In other words, producing the acceptable effects of cmd, next the acceptable effects of $E\langle v\rangle$ (with v the result value of cmd) and a result in P' should be acceptable in $\mu\ P'$. The third and final axiom A-PRINT makes the link between μ and ρ by requiring that an effect produced using acceptable values (including primitive capabilities in ρ) should be acceptable: if $v \in Val_{\mu,\rho}$, then $v.\texttt{print}(v')$ is in $\mu\ Val_{\mu,\rho}$ for any value v'.

These admissibility axioms impose natural conditions for properties to be (in principle) enforceable on untrusted code using the object capability approach. Axiom A-PURE models the fact that we cannot prevent untrusted code from returning any valid result value that it chooses. Similarly, Axiom A-BIND means that we cannot prevent untrusted code from composing two expressions that we deem acceptable individually. Finally, Axiom A-PRINT means that we cannot prevent the code from exercising the primitive capabilities that we allow it to access.

We point out that Axiom A-BIND should not be interpreted to mean that we cannot enforce policies that are stateful. For example, when we add mutable state in Section 3.1, a policy that only one output may happen can be modelled by using a flag heap variable and requiring that output can only happen when the flag is 0 and after outputting anything, it should be set to 1.

Scope Our third and final step towards formulating effect parametricity, requires formalising the values that an expression has access to. We use a notion of well-scopedness of expressions. Figure 2 defines contexts Γ as lists of variables and a judgement $\Gamma \vdash e$ expressing that e only uses variables in Γ. We define $[\![\Gamma]\!]_{\mu,\rho}$ as the set of substitutions γ that map the variables in Γ to acceptable values in $Val_{\mu,\rho}$:

$$[\![\Gamma]\!]_{\mu,\rho} \stackrel{\mathsf{def}}{=} \{\gamma \mid \gamma(x) \in Val_{\mu,\rho} \text{ for all } x \in \Gamma\}.$$

We can now formulate our Fundamental Theorem.

$e ::= \cdots \mid \lambda x.e \mid e\ e \qquad v ::= \cdots \mid \lambda x.e \qquad E ::= \cdots \mid E\ e \mid v\ E$

$$(\lambda x.e_1)v_2 \hookrightarrow e_1[x/v_2] \qquad \text{(E-APP)}$$

Figure 3. $\lambda_{\texttt{out},HO}$, a higher-order calculus with an output capability.

Theorem 1 (Fundamental Theorem for $\lambda_{\texttt{out},FO}$)**.** *For a valid effect interpretation (μ, ρ), $\Gamma \vdash e$ implies that for any $\gamma \in [\![\Gamma]\!]_{\mu,\rho}$, $\gamma(e)$ is in $\mathcal{E}[\mu\ Val_{\mu,\rho}]$.*

The theorem states that for an arbitrary expression e with access to variables in Γ only (i.e. $\Gamma \vdash e$), instantiating those variables with acceptable values (i.e. $\gamma \in [\![\Gamma]\!]_{\mu,\rho}$) produces an expression $\gamma(e)$ with acceptable effects and result value: $\gamma(e) \in \mathcal{E}[\mu\ Val_{\mu,\rho}]$. A proof is in the TR, by structural induction on the well-scopedness judgement $\Gamma \vdash e$.

Unfortunately, $\lambda_{\texttt{out},FO}$ is too simple for the theorem to imply many useful results. Without function values or objects, there is nothing to play the role of an instance object: a value that reacts in a programmer-defined way to some form of (method) invocation. This rules out custom capabilities, and the only result to prove is that code without a reference to primitive capability out cannot produce output. It is still instructive to see how that follows.

Example 1. *If $\Gamma \vdash e$, $\gamma(x) \neq \texttt{out}$ for all x in Γ and $\gamma(e) \rightarrow_o^* v$. Then o must be empty.*

Proof. Instantiate Theorem 1 with effect interpretation $(\mu, \rho) = (IO_{triv}, Ref_{triv})$, where $Ref_{triv} \stackrel{\mathsf{def}}{=} \emptyset$ and

$$IO_{triv}\ P \stackrel{\mathsf{def}}{=} \{cmd \mid \text{If } cmd \rightarrow_o^* v \text{ then } o = [] \wedge v \in P\}$$

Ref_{triv} defines that there are *no* acceptable primitive capabilities and $IO_{triv}\ P$ declares a command acceptable if evaluating it produces *no* output and a result satisfying P. In the TR, we show that (IO_{triv}, Ref_{triv}) is a valid effect interpretation. With $\Gamma \vdash e$ and $\gamma \in [\![\Gamma]\!]_{IO_{triv},Ref_{triv}}$, Theorem 1 implies that $\gamma(e) \in \mathcal{E}[IO_{triv}\ Val_{IO_{triv},Ref_{triv}}]$. $\gamma(e) \rightarrow_o^* v$ then implies (by a simple lemma) that $o = []$. ☐

This result is not useless, but probably easier to prove without our machinery. In a higher-order calculus, *custom* capabilities make the Fundamental Theorem more useful.

2.2. An output capability in a higher-order calculus

To obtain a higher-order calculus, we add standard first-class functions in Figure 3, producing calculus $\lambda_{\texttt{out},HO}$.

The definition of commands does not change and we reuse command predicates and $\mathcal{E}[P]$ as defined before. An effect interpretation is still a couple (μ, ρ), with $\mu \in \mathcal{P}(Val) \rightarrow \mathcal{P}(Cmd)$ and $\rho \in \mathcal{P}(Cap)$. However, the set of acceptable values w.r.t. effect interpretation (μ, ρ) becomes more complicated, since lambdas are values too. We define a command predicate $P_1 \rightarrow P_2$ containing lambdas that map values in predicate P_1 to expressions in $\mathcal{E}[P_2]$.

$$P_1 \rightarrow P_2 \stackrel{\mathsf{def}}{=} \{\lambda x.e \mid \text{For all } v \in P_1, e[x/v] \text{ is in } \mathcal{E}[P_2]\}$$

It is natural to extend $Val_{\mu,\rho}$ with those lambdas that produce acceptable effects and acceptable results when invoked with an acceptable argument, i.e. lambdas in $Val_{\mu,\rho} \to \mu\ Val_{\mu,\rho}$:

$$Val_{\mu,\rho} \overset{\mathsf{def}}{=} PureVal \cup \rho \cup \big(Val_{\mu,\rho} \to \mu\ \ Val_{\mu,\rho}\big) \qquad \text{(Oops!)}$$

Unfortunately, the resulting definition is recursive in an unacceptable way. It defines $Val_{\mu,\rho}$ recursively in terms of itself and this is mathematically unsound[3]. Formally, the recursive equation may not have a solution (a value predicate that satisfies the equation). Luckily, we are not the first to encounter this problem. A well-studied technique called *step-indexing* can be used to solve it [13].

The idea is to no longer work with command predicates but use *step-indexed* command predicates instead. These are predicates on couples of natural numbers and commands: $P \in \mathcal{P}(\mathbb{N} \times Cmd)$. The fact that a couple (n, cmd) is in a step-indexed predicate P can be understood as saying that cmd will satisfy P when we inspect it for a maximum of n steps. We say that cmd is n-acceptable in P. For example, for noticing that the expression $e = \mathtt{let}(y = 3)\ \ \mathtt{let}\ (z = 4)\ \mathtt{true}$ produces a boolean, we need to perform 2 computational steps. Therefore, $(0, e)$ and $(1, e)$ could be in $\mathcal{E}[P]$ regardless of P but $(2, e)$ only if $\mathtt{true}$ is considered acceptable by P. Typically, we require for step-indexed predicates that they are *uniform*; inspecting expressions for more steps should only make more expressions unacceptable. More formally, we define $UPred(A)$, the set of uniform step-indexed predicates over a set A as follows. For technical reasons, we sometimes also use normal predicates in $Pred(A)$.

$$Pred(A) \overset{\mathsf{def}}{=} \mathcal{P}(\mathbb{N} \times A)$$
$$UPred(A) \overset{\mathsf{def}}{=} \left\{ \begin{array}{l} p \in \mathcal{P}(\mathbb{N} \times A) \mid \\ \quad (n, e) \in p \text{ and } k \leq n \text{ implies that } (k, e) \in p \end{array} \right\}$$

We adapt our definitions to follow suit:

$$\mathcal{E}[P] \overset{\mathsf{def}}{=} \left\{ (k, e) \;\middle|\; \begin{array}{r} \text{For all } cmd \text{ and } i \leq k \text{ s.t. } e \to^{i} cmd, \\ \text{we have that } (k - i, cmd) \in P \end{array} \right\}$$

That is: an expression is k-acceptable in the expression extension of command predicate P if a command that it evaluates to in $i \leq k$ steps is $(k - i)$-acceptable in P.

Effect interpretations become couples (μ, ρ) with $\mu \in UPred(Val) \to Pred(Cmd)$ and $\rho \in UPred(Cap)$ and we can now correct the failed definition of $Val_{\mu,\rho}$:

$$P_1 \to P_2 \overset{\mathsf{def}}{=} \left\{ (n, \lambda x.e) \;\middle|\; \begin{array}{r} \text{For all } i < n, \text{ and } (i, v) \in P_1, \\ \text{we have that } (i, e[x/v]) \in \mathcal{E}[P_2] \end{array} \right\}$$
$$Val_{\mu,\rho} \overset{\mathsf{def}}{=} (\mathbb{N} \times PureVal) \cup \rho \cup \big(Val_{\mu,\rho} \to \mu\ \ Val_{\mu,\rho}\big)$$

The predicate of functions $P_1 \to P_2$ defines as n-acceptable those lambdas that, for $i < n$, can be given an i-acceptable argument to obtain an i-acceptable result.

Thanks to step-indexing, this definition of $Val_{\mu,\rho}$ is mathematically valid. Intuitively, this is because deciding whether or not $\lambda x.e$ is n-accepted by $P_1 \to P_2$, only requires knowing P_1 and P_2 up to $n - 1$ steps. Hence, whether an expression is n-acceptable in $Val_{\mu,\rho}$ only depends on the $(n - 1)$-acceptable expressions in $Val_{\mu,\rho}$. The more mathematical story is that with a certain *metric* (a function that defines a "distance" between two predicates), $UPred(A)$ can be seen as a *complete metric space* and the Banach fixpoint theorem guarantees that unique fixpoints exist for *contractive* functions (i.e. applying the function to two predicates produces new predicates that are strictly closer together than the original two). The TR explains this more formally.

3. The contravariant occurrence of $Val_{\mu,\rho}$ precludes induction.

Valid effect interpretations (μ, ρ) must satisfy natural adaptations of the axioms we saw before.

- A-PURE: For $(n, v) \in P$, (n, v) must be in $\mu\ P$.
- A-BIND: If $(n, cmd) \in \mu\ P$ and $(i, E\langle v\rangle) \in \mathcal{E}[\mu\ P']$ for all $i \leq n$ and values $(i, v) \in P$, then $(n, E\langle cmd\rangle) \in \mathcal{E}[\mu\ P']$.
- A-PRINT: If $(n, v) \in Val_{\mu,\rho}$, then $(n, v.\mathtt{print}(v')) \in \mu\ \ Val_{\mu,\rho}$.

Context interpretations now become step-indexed as well: $(n, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}$ iff $(n, \gamma(x)) \in Val_{\mu,\rho}$ for all $x \in \Gamma$. Well-scopedness is easily extended to lambdas and applications:

$$\frac{\Gamma, x \vdash e}{\Gamma \vdash \lambda x.e} \qquad\qquad \frac{\Gamma \vdash e_1 \qquad \Gamma \vdash e_2}{\Gamma \vdash e_1\ e_2}$$

We can now state the Fundamental Theorem for $\lambda_{\mathtt{out},HO}$.

Theorem 2 (Fundamental Theorem for $\lambda_{\mathtt{out},HO}$)**.** *For a valid effect interpretation* (μ, ρ)*,* $\Gamma \vdash e$ *implies for any* n *and* $(n, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}$ *that* $(n, \gamma(e))$ *is in* $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$*.*

This theorem naturally adapts the previous theorem to step-indexing and as before, it formalises our intuitive notion of effect parametricity; if an expression e only has access to variables in Γ and these are instantiated by values that produce acceptable effects by effect interpretation (μ, ρ), then the resulting expression $\gamma(e)$ produces effects acceptable by (μ, ρ) and a result that can only produce acceptable effects.

The lambdas in $\lambda_{\mathtt{out},HO}$ can be used as custom capabilities. Assume a primitive function $toUpperCase$ that converts strings to uppercase and consider the following expression:

$$upp \overset{\mathsf{def}}{=} \lambda s.\,\mathtt{out.print}(toUpperCase(s))$$

The function upp converts a string to upper case and prints it using primitive capability $\mathtt{out}$. The expression is a custom capability, restricting $\mathtt{out}$ to allow printing only uppercase text. But can we prove that this restriction actually holds?

Example 2. *For expression* e *and variable* u*, if* $\emptyset, u \vdash e$ *and* $e[u \mapsto upp] \to_{o}^{*} v$*, then the output* o *is in uppercase.*

Proof. Define effect interpretation (IO_{upp}, Ref_{upp}):

$$Ref_{upp} \overset{\mathsf{def}}{=} \emptyset$$
$$IO_{upp}\ \ P \overset{\mathsf{def}}{=} \left\{ (n, cmd) \;\middle|\; \begin{array}{r} \text{For } i \leq n, \text{ if } cmd \to_{o}^{i} v \text{ then } o \text{ is} \\ \text{in upper case and } (n - i, v) \in P \end{array} \right\}$$

Ref_{upp} defines that $\mathtt{out}$ is not an acceptable primitive capability (upp is only useful when no direct access to $\mathtt{out}$ is available). IO_{upp} defines as n-acceptable those expressions that, when evaluated to a value in $i \leq n$ steps, produce only uppercase output and an $n - i$-acceptable result. In the TR, we show that (IO_{upp}, Ref_{upp}) is a valid effect interpretation.

Although out is not in ρ, (n, upp) is in $Val_{IO_{upp}, Ref_{upp}}$ for any n, so that $(n, [u \mapsto upp])$ is in $[\![u, \emptyset]\!]_{IO_{upp}, Ref_{upp}}$. This follows because (n, upp) is in

$$\left(Val_{IO_{upp}, Ref_{upp}} \to IO_{upp}\ \ Val_{IO_{upp}, Ref_{upp}}\right) \subseteq Val_{IO_{upp}, Ref_{upp}}.$$

For $i < n$ and $(i, v) \in Val_{IO_{upp}, Ref_{upp}}$, we show

$$(i, \texttt{out.print}(toUpperCase(v))) \in \mathcal{E}[IO_{upp}\ \ Val_{IO_{upp}, Ref_{upp}}]$$

So, take $i' \leq i$ and $\texttt{out.print}(toUpperCase(v)) \to^{i'} cmd$. Then i' must be 1 and cmd must be $\texttt{out.print}(v')$ for v' the uppercased version of v. We show that $\texttt{out.print}(v')$ is $i-1$-acceptable in $IO_{upp}\ \ Val_{IO_{upp}, Ref_{upp}}$, so take $i'' \leq i - 1$ and $\texttt{out.print}(v') \to_{o}^{i''} v''$. Then $i'' = 1$, $v'' = \texttt{undef}$ and $o = [v']$. $(i-2, \texttt{undef})$ is in $Val_{IO_{upp}, Ref_{upp}}$ and o is in upper case.

The Fundamental Theorem then implies the safety of upp. For arbitrary n, (n, upp) is in $Val_{IO_{upp}, Ref_{upp}}$ and thus $(n, [u \mapsto upp])$ is in $[\![\emptyset, u]\!]_{IO_{upp}, Ref_{upp}}$. (IO_{upp}, Ref_{upp}) is a valid effect interpretation, so effect parametricity implies that $(n, e[u \mapsto upp])$ is in $\mathcal{E}[IO_{upp}\ \ Val_{IO_{upp}, Ref_{upp}}]$. By a simple lemma in the TR, this gives the required result. □

2.3. Dealing with ambient authority

To understand effect parametricity, it is instructive to consider how our proofs would fail for non-capability-safe languages. Remember, for example, that out is *internal* syntax, not *surface* syntax, i.e. the programmer is not allowed to write it. Formally, our well-scopedness judgement $\Gamma \vdash e$ does not permit e to mention out, so that our Fundamental Theorem does not apply to expressions that do mention out.

If we drop this restriction, i.e. make out part of the surface syntax, all programs gain implicit or *ambient authority* to produce arbitrary output. The custom capability upp no longer works: programs can just use the stronger capability out instead of upp to produce arbitrary output. Formally, if we make out part of the surface syntax by adding the rule $\Gamma \vdash \texttt{out}$ to the well-scopedness judgement, then the Fundamental Theorem should additionally require (as an axiom) that the effect interpretation considers out acceptable, i.e. $(n, \texttt{out}) \in \rho$ for any n. This excludes the interpretation used to prove safety of upp, but since upp is no longer safe, this makes sense.

We emphasise however, that the additional ambient authority *can* be accommodated with some changes to our Fundamental Theorem. The benefit is small for $\lambda_{\texttt{out},HO}$, since an effect interpretation allowing out cannot impose any restriction on effects at all, but imagine that there were primitive capabilities beside out. With a non-globally accessible net capability for accessing the network, for example, the modified Fundamental Theorem still implies properties about how the network can be accessed and custom capabilities restricting network access can still work, despite the ambient authority of the globally accessible out.

3. Capability-safety in λ_{JS}

In this paper, we present our results for λ_{JS} (although we believe they can be adapted to other object capability languages, both typed and untyped, both low-level and high-level). The higher-order store in λ_{JS} adds significant complexity but makes the results more realistic and interesting.

3.1. LambdaJS

Figure 4 shows the syntax and operational semantics of λ_{JS}, as defined by Guha et al. [15], but omitting exceptions and object prototypes. The calculus is a fairly standard untyped lambda calculus with numbers, strings, booleans, undef and null values. There are n-ary lambdas as func expressions and string-indexed records with field projection $e[e]$, field update $e[e] = e$, field deletion $\texttt{delete}\ e[e]$ and record literals $\{str : e\}$. The record operations are pure; for example, a field update $r[\text{“}fld\text{”}] = 5$ does not modify r, but returns a modified copy. Furthermore, λ_{JS} has mutable references with update $(e_1 = e_2)$, allocation ($\texttt{ref}\ e$; allocates a memory cell with initial value e) and dereference ($\texttt{deref}\ e$) expressions. Finally, there are normal if, sequencing $(e; e)$ and while expressions and unspecified primitive operators $\texttt{op}_n$.

The operational semantics of λ_{JS} is defined in two parts. The primitive pure evaluation judgement $e_1 \hookrightarrow e_2$ defines the evaluation of func expressions, field projections, field updates, field deletions, in addition to rules for let, if, sequencing expressions and while expressions in Figure 1. Primitive operations op_n evaluate in terms of an unspecified δ_n function. The actual small-step pure evaluation judgement $e_1 \to e_2$ is defined in terms of $e_1 \hookrightarrow e_2$ and evaluation contexts, obtaining a strict, left-to-right evaluation order. Finally, the impure evaluation judgement $(\sigma_1, e_1) \to (\sigma_2, e_2)$ for stores σ embeds pure evaluations, leaving the store unmodified, and defines the behaviour of allocation ($\texttt{ref}\ e$), dereference ($\texttt{deref}\ e$) and assignment $(e = e)$ expressions.

Like $\lambda_{\texttt{out},FO}$ and $\lambda_{\texttt{out},HO}$, our subset of λ_{JS} has many stuck terms, e.g., an if branching on a non-boolean or invoking a non-func as a function. The original λ_{JS} produced exceptions in those cases, but we omit those for simplicity.

3.2. The logical relation

Let us now extend our previous results to λ_{JS}. As for $\lambda_{\texttt{out},HO}$, we use step-indexing to construct them in a mathematically sound way. However, the higher-order store that is present in λ_{JS} adds significant complexity. It adds new types of authority (accessing and modifying heap data structures, respecting certain invariants or protocols, in arbitrary or restricted ways) but also new ways for components to communicate and pass capabilities to each other.

To support all of this, we construct additional machinery to track assumptions that components have about shared data structures and the authority they have to modify them. We construct a relatively rich type of *Kripke worlds* to model arbitrary invariants and protocols on shared state [14], [17].

$$
\begin{array}{llll}
l \in \mathit{Loc} & c \in \mathit{Const} ::= \mathit{num} \mid \mathit{str} \mid \mathit{bool} \mid \texttt{undef} \mid \texttt{null} & \sigma \in \mathit{Store} ::= (l, v) \cdots & \delta_n : \mathit{op}_n \times v_1 \cdots v_n \to c
\end{array}
$$

$$
\begin{array}{rl}
v \in \mathit{Val} ::= & c \mid \texttt{func}(x \cdots)\{\texttt{return } e\} \mid \{\mathit{str} : v\} \mid l \\
e \in \mathit{Expr} ::= & x \mid v \mid \texttt{let}(x = e)\ e \mid e(e \cdots) \mid e[e] \mid e[e] = e \mid \texttt{delete } e[e] \mid \{\mathit{str} : e\} \mid e = e \mid \texttt{ref } e \mid \texttt{deref } e \\
\mid & \texttt{if}(e)\{e\}\ \texttt{else}\ \{e\} \mid e; e \mid \texttt{while}(e)\{e\} \mid \mathit{op}_n(e_1 \cdots e_n) \\
E ::= & \cdot \mid \texttt{let}(x = E)\ e \mid E(e \cdots) \mid v(v \cdots E, e \cdots) \mid \{\mathit{str} : v \cdots, \mathit{str} : E, \mathit{str} : e \cdots\} \mid E[e] \mid v[E] \mid E[e] = e \mid v[E] = e \mid v[v] = E \\
\mid & \texttt{delete } E[e] \mid \texttt{delete } v[E] \mid E = e \mid v = E \mid \texttt{ref } E \mid \texttt{deref } E \mid \texttt{if}(E)\{e\}\ \texttt{else}\ \{e\} \mid E; e \mid \mathit{op}_n(v \cdots E\ e \cdots)
\end{array}
$$

$$
\texttt{func}(x_1 \cdots x_n)\{\texttt{return } e\}(v_1 \cdots v_n) \hookrightarrow e[x_1/v_1 \cdots x_n/v_n] \quad \text{(E-APP)} \qquad \frac{\mathit{str}_x \notin (\mathit{str}_1 \cdots)}{\{\mathit{str}_1 : v_1 \cdots\}[\mathit{str}_x] = v_x \hookrightarrow \{\mathit{str}_x : v_x, \mathit{str}_1 : v_1 \cdots\}} \quad \text{(E-CREATEFIELD)}
$$

$$
\frac{\mathit{str}_x \notin (\mathit{str}_1 \cdots)}{\texttt{delete } \{\mathit{str}_1 : v_1 \cdots\}[\mathit{str}_x] \hookrightarrow \{\mathit{str}_1 : v_1 \cdots\}} \quad \text{(E-DELETEFIELD-NOTFND)} \qquad \frac{\mathit{str}_x \notin (\mathit{str}_1 \cdots \mathit{str}_n)}{\{\mathit{str}_1 : v_1 \cdots \mathit{str}_n : v_n\}[\mathit{str}_x] \hookrightarrow \texttt{undef}} \quad \text{(E-GETFIELD-NOTFND)}
$$

$$
\{\cdots \mathit{str} : v \cdots\}[\mathit{str}] \hookrightarrow v \quad \text{(E-GETFIELD)} \qquad \mathit{op}_n(v_1 \cdots v_n) \hookrightarrow \delta_n(\mathit{op}_n, v_1 \cdots v_n) \quad \text{(E-PRIM)}
$$

$$
\{\mathit{str}_1 : v_1 \cdots \mathit{str}_i : v_i \cdots \mathit{str}_n : v_n\}[\mathit{str}_i] = v \hookrightarrow \{\mathit{str}_1 : v_1 \cdots \mathit{str}_i : v \cdots \mathit{str}_n : v_n\} \quad \text{(E-UPDATEFIELD)}
$$

$$
\texttt{delete } \{\mathit{str}_1 : v_1 \cdots \mathit{str}_x : v_x \cdots \mathit{str}_n : v_n\}[\mathit{str}_x] \hookrightarrow \{\mathit{str}_1 : v_1 \cdots \mathit{str}_n : v_n\} \quad \text{(E-DELETEFIELD)}
$$

Define $e_1 \to e_2$ if $e_1 = E\langle e_1' \rangle$, $e_2 = E\langle e_2' \rangle$ and $e_1' \hookrightarrow e_2'$.

$$
\frac{e_1 \hookrightarrow e_2}{(\sigma, E\langle e_1 \rangle) \to (\sigma, E\langle e_2 \rangle)} \quad \text{(E-PURE)} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{(\sigma, E\langle \texttt{ref } v \rangle) \to (\sigma[l \mapsto v], E\langle l \rangle)} \quad \text{(E-REF)} \qquad
\begin{array}{ll}
(\sigma, E\langle \texttt{deref } l \rangle) \to (\sigma, E\langle \sigma(l) \rangle) & \text{(E-DEREF)} \\
(\sigma, E\langle l = v \rangle) \to (\sigma[l \mapsto v], E\langle v \rangle) & \text{(E-SETREF)}
\end{array}
$$

Figure 4. Syntax and operational semantics of λ_{JS}, following Guha et al. [15], but omitting exceptions and object prototypes. For brevity, we write `undef` instead of `undefined` and sometimes omit brackets and the `return` in `func`s and we write (σ, e) instead of σe for clarity. The figure defines values v, expressions e and evaluation contexts E and the primitive pure evaluation judgement $e \hookrightarrow e$ (not repeating rules E-WHILE, E-LET, E-IFTRUE, E-IFFALSE and E-BEGINDISCARD from Figure 1), the pure evaluation judgement $e \to e$ and the impure evaluation judgement $(\sigma, e) \to (\sigma, e)$.

Our notations loosely follow Birkedal et al. [16] and we use their recipe for constructing recursive worlds (see below).

Kripke possible worlds

λ_{JS} features a higher-order mutable store, i.e., one can use references into *heap* memory that may contain higher-order data such as functions or objects. It is important that our logical relations are powerful enough to support typical usage of such references, and this is quite a challenge. Often, references into the store are shared between multiple components and correctness of a program relies upon invariants on their contents. For example, *rnode* in the introduction needs to know that the DOM is a tree, not a graph. Sometimes invariants evolve during execution, e.g., an auction object may contain a list of bids that is modifiable, but not after the auction is marked final. Components may also have partial authority over a shared data structure. For example, the ad in the example from the introduction only has the authority to modify its part of the DOM, while other components may modify the whole DOM. Similarly, Section 4.3 studies two isolated components, one of which can only push values on a stack while the other can only pop.

To support such patterns, we use another well-established solution: *Kripke* logical relations. The idea is to index our predicates by *possible worlds* $w \in W$, which model a set of assumptions about (a) the current state of data structures in the store, (b) invariants and protocols that will be respected over them in the future and (c) the authority that is available to modify those data structures. Intuitively, a value v is n-accepted by an indexed predicate $P \in W \to \mathit{UPred}(\mathit{Val})$ in a world w (i.e. $(n, v) \in P\ w$) when inspecting v for n operational steps in stores satisfying w cannot make it break invariants in w or return an invalid result. For values (which can be stored and used later), we will require that they are valid not only in w, but in any possible future evolution of $w' \sqsupseteq w$ (to be defined later).

We define our Kripke worlds as follows:

$$
\begin{aligned}
&\mathrm{IslandName} \overset{\mathsf{def}}{=} \mathbb{N} \\
&W \overset{\mathsf{def}}{=} \{ w \in \mathrm{IslandName} \hookrightarrow \mathrm{Island} \mid \mathrm{dom}(w) \text{ finite} \} \\
&\mathrm{Island} \overset{\mathsf{def}}{=} \left\{ \begin{array}{c} \iota = (s, \phi, \phi^{\mathrm{pub}}, H) \mid s \in \mathrm{State} \wedge \phi \subseteq \mathrm{State}^2 \wedge \\ H \in \mathrm{State} \to \mathrm{StorePred} \wedge \phi^{\mathrm{pub}} \subseteq \phi \wedge \\ \phi, \phi^{\mathrm{pub}} \text{ reflexive and transitive} \end{array} \right\} \\
&\mathrm{StorePred} \overset{\mathsf{def}}{=} \{ \psi \in \hat{W} \to_{mon,ne} \mathit{UPred}(\mathrm{Store}) \} \\
&\mathit{roll} : \frac{1}{2} \cdot W \cong \hat{W}
\end{aligned}
$$

Worlds $w \in W$ contain a finite set of components or *islands* describing disjoint parts of the store. Islands are identified by $\mathrm{IslandName}$s which are in fact just natural numbers. An island ι represents an evolvable invariant about a heap data structure, as well as a lower bound on the available authority to modify the data structure. Concretely, $\iota = (s, \phi, \phi^{\mathrm{pub}}, H)$ represents a state machine with current state s and transition relation $\phi \in \mathrm{State}^2$. We assume a fixed set State of possible states, assumed to contain all the states used in this paper.

Every island contains a function $H \in \mathrm{State} \to \mathrm{StorePred}$ that defines when a store satisfies the requirements for a state. These store requirements are modelled as $\mathrm{StorePred}$s: predicates on stores that are themselves again world-indexed. The arrow $\to_{mon,ne}$ means that H must be *monotone* (to be explained later) as well as non-expansive (a sanity requirement w.r.t. step-indexing, see the TR).

Finally, ϕ^{pub} represents an assumption about the available authority to modify the data structure. It is a subrelation of ϕ, and it represents a subset of state machine transitions for which authority is available to make. For example, in Section 4.2, the ad example from the introduction will be verified using an island whose states are trees of values that represent the current state of the DOM. We will prove that the *ad.initialize* call is valid w.r.t. a world in which

ϕ^{pub} only allows modifications in the ad's part of the tree (since the ad may not modify the DOM outside of its node), while ϕ allows arbitrary modifications to the DOM (since the DOM may be modified arbitrarily by other code).

The careful reader may notice that the above definition of worlds is again recursive: worlds contain store predicates, which are themselves indexed by worlds. This kind of recursive worlds is needed to deal with the higher-order nature of λ_{JS} and we use a general method by Birkedal et al. [16] for constructing them. The factor $\frac{1}{2}$, the set $\hat{W}$ and the isomorphism *roll* play a technical role in this construction, but we recommend the casual reader to ignore them here and elsewhere in the paper.

Future worlds Kripke worlds represent assumptions that a piece of code holds on the rest of the system. However, because values may be stored and used later, any assumptions that they rely on must not be invalidated by legitimate future evolutions of the system. There are essentially three legitimate ways for a system to evolve. (a) Fresh data structures may be allocated, and invariants or protocols may be established for them. Additionally, (b) existing invariants may evolve according to their established protocols and finally (c) additional authority over data structures may become available. These three types of evolutions are modelled by the future world relation; a world w_2 is a future world of w_1 ($w_2 \sqsupseteq w_1$) if w_2 contains at least the islands of w_1, but potentially more (a). For existing islands, the state in w_2 must be reachable from the state in w_1 using the state machine transitions in ϕ, i.e. according to the data structure's established protocol (b). Finally, the state machine's set of public transitions ϕ^{pub} is allowed to grow (but not beyond its complete set of transitions ϕ), representing an increase in available authority (c).

$$w_2 \sqsupseteq w_1 \text{ iff } \operatorname{dom}(w_2) \supseteq \operatorname{dom}(w_1) \wedge \forall j \in \operatorname{dom}(w_1).\ w_2(j) \sqsupseteq w_1(j)$$

$$(s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2) \sqsupseteq (s_1, \phi_1, \phi_1^{\mathrm{pub}}, H_1) \text{ iff}$$
$$(\phi_2, H_2) = (\phi_1, H_1) \text{ and } \phi_1 \supseteq \phi_2^{\mathrm{pub}} \supseteq \phi_1^{\mathrm{pub}} \text{ and } (s_1, s_2) \in \phi_1$$

The requirement that values that are valid in a predicate remain valid in legitimate future evolutions of a system is captured by the required *monotonicity* of predicates in $W \to_{mon,ne} \mathit{UPred}(A)$ (in addition to non-expansiveness, a sanity requirement w.r.t. step-indexing). Formally, monotonicity requires that $P\ w_2 \supseteq P\ w_1$ whenever $w_2 \sqsupseteq w_1$, i.e. values/stores/... that are valid in a world w_1 must remain valid in future worlds $w_2 \sqsupseteq w_1$. In what follows, predicates on values and stores will typically need to be monotone (because they contain values that may be stored and used later) while predicates on commands and expressions typically need not be, as they cannot be stored (except as part of a `func` value).

The future world relation $w_2 \sqsupseteq w_1$ represents future evolutions of a world that a piece of code should be able to cope with. However, it is not necessarily allowed to make those changes itself. A more restricted *public* future-world relation $w_2 \sqsupseteq^{\mathrm{pub}} w_1$ defines the set of future worlds that one has the authority to transition to:

$$w_2 \sqsupseteq^{\mathrm{pub}} w_1 \text{ iff } \begin{cases} \operatorname{dom}(w_2) \supseteq \operatorname{dom}(w_1) \wedge \\ \quad \forall j \in \operatorname{dom}(w_1).\ w_2(j) \sqsupseteq^{\mathrm{pub}} w_1(j) \end{cases}$$

$$(s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2) \sqsupseteq^{\mathrm{pub}} (s_1, \phi_1, \phi_1^{\mathrm{pub}}, H_1) \text{ iff}$$
$$(\phi_2, \phi_2^{\mathrm{pub}}, H_2) = (\phi_1, \phi_1^{\mathrm{pub}}, H_1) \text{ and } (s_1, s_2) \in \phi_1^{\mathrm{pub}}$$

A public future world $w_2 \sqsupseteq^{\mathrm{pub}} w_1$ must also contain at least the islands of w_1 and may contain additional ones. However, the new state of islands in w_2 must now be reachable through the state machine's public transitions in ϕ_1^{pub}, i.e. transitions that can be made using the available authority. Finally, ϕ^{pub} is not allowed to grow in public future worlds, i.e. one does not have the authority to increase one's own authority. Note that the latter precludes capabilities appearing out of thin air, somewhat similar to the *No Authority Amplification* property that we will discuss in Section 5.

Disjoint worlds (with assumptions about disjoint data structures in the heap) can be combined with the $\oplus$ operator:

$$w \oplus w' = w'' \text{ iff } \operatorname{dom}(w'') = \operatorname{dom}(w) \uplus \operatorname{dom}(w') \wedge \forall j \in \operatorname{dom}(w).$$
$$w''(j) = w(j) \wedge \forall j \in \operatorname{dom}(w').\ w''(j) = w'(j)$$

Finally, a store σ n-satisfies a world w if it can be partitioned into parts that n-satisfy the store predicate for the current state of all islands (we write $\iota.H$ for projecting out the store predicate of an island):

$$\sigma :_n w \text{ iff } \begin{cases} \exists \sigma_j. \sigma = \uplus_{j \in \operatorname{dom}(w)} \sigma_j \text{ and } \forall j \in \operatorname{dom}(w), \\ \quad \forall n' < n.\ (n', \sigma_j) \in w(j).H(w(j).s)\ (\mathit{roll}\ w) \end{cases}$$

It is worthwhile at this point to take a step back and build an intuitive understanding for our worlds. Generally, they should be interpreted as a set of assumptions with respect to which a value or expression is valid. An expression e that holds capabilities for making certain modifications to a shared data structure, will only be valid w.r.t. a Kripke world with an island governing the data structure. Private transitions for the island would include all possible modifications to the data structure that *any* subject in the system has the authority to make, so that e will be required to tolerate such modifications. Public transitions for the island will include at least the modifications that e itself has the authority to make. Using security terminology, this perspective means that any expression may be seen as a subject and the world in which it is valid as an upper bound on its required authority. Entities in our approach are simply represented by code executing on their behalf.

Command predicates As for $\lambda_{\mathtt{out},FO}$ and $\lambda_{\mathtt{out},HO}$, we define a syntactic class of *commands*: either values or expressions blocked on an impure operation.

$$\mathit{cmd}_0 ::= \mathtt{deref}\ v \mid v = v \mid \mathtt{ref}\ v \qquad \mathit{cmd} \in \mathit{Cmd} ::= E\langle \mathit{cmd}_0 \rangle \mid v$$

Predicates on commands P are again extended to expressions as $\mathcal{E}[P]$, which closes P under pure evaluation:

$$\mathcal{E}[P] : W \to_{ne} \mathit{Pred}(\mathit{Expr})$$

$$\mathcal{E}[P]\ w \stackrel{\mathsf{def}}{=} \left\{ (n, e) \;\middle|\; \begin{array}{l} \text{for all } i \leq n, e'. \text{if } e \to^i \mathit{cmd}, \\ \quad \text{then } (n - i, \mathit{cmd}) \in P\ w \end{array} \right\}$$

A few predicates are used as building blocks further on:

$$Cnst : W \rightarrow_{mon,ne} UPred(Val)$$
$$Cnst\ w \overset{\text{def}}{=} \mathbb{N} \times Const$$
$$\{P\} : W \rightarrow_{mon,ne} UPred(Val)$$
$$\{P\}\ w \overset{\text{def}}{=} \left\{(n, \{\overline{str : v}\}) \mid \text{for all } i < n, \overline{(i, v) \in P\ w}\right\}$$
$$P \cup P' : W \rightarrow_{mon,ne} UPred(Val)$$
$$(P \cup P')\ w \overset{\text{def}}{=} P\ w \cup P'\ w$$
$$([P] \rightarrow P'') : W \rightarrow_{mon,ne} UPred(Val)$$
$$([P] \rightarrow P'')\ w \overset{\text{def}}{=} \left\{(n, \texttt{func}(x_1 \cdots x_k)\{\texttt{return } e\}) \mid \begin{array}{r} \text{for all } v_1 \cdots v_k, w' \sqsupseteq w, i < n.\ \overline{(i, v_j) \in P\ w'} \Rightarrow \\ (i, e[x_1/v_1, \cdots, x_n/v_k]) \in \mathcal{E}[P'']\ w' \end{array}\right\}$$

$Cnst$ n-accepts all constant values. For value predicates P and P' and command predicate P'', $\{P\}$ accepts records with P-acceptable fields and $P \cup P'$ accepts values from P or P'. $[P] \rightarrow P''$ n-accepts `func` expressions producing i-acceptable results in P'' when applied to i-acceptable arguments in P, in any future world $w' \sqsupseteq w$ and for any $i < n$. As a technical detail, the quantification over $w' \sqsupseteq w$ and $i < n$ makes $[P] \rightarrow P''$ monotone and uniform even when P'' is not.

Effect interpretations and acceptable values Like for $\lambda_{\texttt{out},FO}$ and $\lambda_{\texttt{out},HO}$, we parameterise our LR over an effect interpretation (μ, ρ) with $\rho : W \rightarrow_{mon,ne} UPred(Loc)$ a predicate of references that are valid in a given world and μ a function that maps a predicate on values to a predicate on commands:

$$\mu : (W \rightarrow_{mon,ne} UPred(Val)) \rightarrow_{ne} (W \rightarrow_{ne} Pred(Cmd)).$$

As before, $\mu\ P$ accepts commands producing acceptable effects and results acceptable by a value predicate P.

Given an effect interpretation (μ, ρ) that defines acceptable references and effectful expressions in a given world, we define a predicate $\texttt{JSVal}_{\mu,\rho}$ of all acceptable λ_{JS} values[4]:

$$\texttt{JSVal}_{\mu,\rho} : W \rightarrow_{mon,ne} UPred(Val)$$
$$\texttt{JSVal}_{\mu,\rho} \overset{\text{def}}{=} Cnst \cup \rho \cup \{\texttt{JSVal}_{\mu,\rho}\} \cup ([\texttt{JSVal}_{\mu,\rho}] \rightarrow \mu\ \texttt{JSVal}_{\mu,\rho})$$

The predicate accepts constants, references in ρ, records of acceptable values and functions mapping acceptable values to expressions with acceptable effects and results.

The following axioms are required to hold for a *valid* effect interpretation.

- A-PURE: If $(n, v) \in P\ w$ then $(n, v) \in \mu\ P\ w$
- A-BIND: If $(n, cmd) \in \mu\ P\ w$ and $(n', E\langle v\rangle) \in \mathcal{E}[\mu\ P']\ w'$ for all $n' \leq n$, $w' \sqsupseteq w$ and $(n', v) \in P\ w'$, then $(n, E\langle cmd\rangle) \in \mathcal{E}[\mu\ P']\ w$.
- A-ASSIGN: If $(n, v_1) \in \texttt{JSVal}_{\mu,\rho}\ w$ and $(n, v_2) \in \texttt{JSVal}_{\mu,\rho}\ w$, then $(n, v_1 = v_2) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$.
- A-DEREF: If $(n, v) \in \texttt{JSVal}_{\mu,\rho}\ w$, $(n, \texttt{deref}\ v)$ must be in $\mu\ \texttt{JSVal}_{\mu,\rho}\ w$.
- A-REF: If $(n, v) \in \texttt{JSVal}_{\mu,\rho}\ w$, then $(n, \texttt{ref}\ v) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$.

4. Readers with a background in logical relations can see this as the semantic interpretation of the *unitype* of λ_{JS} values.

Axioms A-PURE and A-BIND are as before (except for the quantification over Kripke worlds) and we don't re-explain them for brevity. Axioms A-ASSIGN and A-DEREF essentially require a compatibility between μ and ρ. They require that if a value is accepted by $\texttt{JSVal}_{\mu,\rho}$ (e.g. references in ρ), then dereferencing it or assigning an acceptable value must be accepted by μ. Finally, Axiom A-REF requires that allocating a new mutable reference (a primitive effect left unrestricted by the language) with an acceptable initial value must be accepted by the effect interpretation.

Fundamental Theorem We conclude this section with the Fundamental Theorem for λ_{JS}: the formal statement of its capability-safety. As before, it informally states that well-formed λ_{JS} terms respect the restrictions on effects imposed by a valid effect interpretation, and now additionally that they respect the invariants and protocols of a Kripke world.

The well-scopedness judgement $\Gamma; \Sigma \vdash e$ states that e is syntactically well-formed in context Γ (a list of free variables) and *store shape* Σ (a list of allocated references). Its definition is unsurprising and relegated to the TR. The predicate $[\![\Sigma]\!]_{\mu,\rho}$ accepts worlds in which the references in Σ are accepted by ρ, and the predicate $[\![\Gamma]\!]_{\mu,\rho}\ w$ accepts substitutions for Γ with acceptable values:

$$[\![\Sigma]\!]_{\mu,\rho} : UPred(W)$$
$$[\![\Sigma]\!]_{\mu,\rho} \overset{\text{def}}{=} \{(n, w) \mid \text{for all } l \in \Sigma. (n, l) \in \rho\ w\}$$
$$[\![\Gamma]\!]_{\mu,\rho}\ w : UPred(Val^{\Gamma})$$
$$[\![\Gamma]\!]_{\mu,\rho}\ w \overset{\text{def}}{=} \{(n, \gamma) \mid \forall x \in \Gamma. (n, \gamma(x)) \in \texttt{JSVal}_{\mu,\rho}\ w\}$$

Theorem 3 (Fundamental Theorem for λ_{JS})**.** *If* $\Gamma, \Sigma \vdash e$ *then for a valid effect interpretation* (μ, ρ) *and for all* n, γ *and* w *with* $(n, w) \in [\![\Sigma]\!]_{\mu,\rho}$ *and* $(n, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w$, *we have that* $(n, \gamma(e))$ *must be in* $\mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$.

The theorem states that substituting acceptable values for an expression's free variables and considering it in a world where its references are acceptable by effect interpretation (μ, ρ), produces a μ-acceptable expression with a result in $\texttt{JSVal}_{\mu,\rho}$. The TR contains a proof by induction on the well-scopedness judgement.

Note that our Fundamental Theorem does not offer termination guarantees about untrusted code. This limitation follows from the higher-order untyped language, in which untrusted code cannot be prevented from diverging.

4. Local state abstraction

A special feature of our logical relation is the quantification over effect interpretations. It allows proving properties about primitive effects, as we saw in Section 2 and we will use it again in Section 6. However, in many cases, standard encapsulation of local state (e.g. instance variables of objects) is all we need. In this section, we define an effect interpretation (IO^{std}, Ref^{std}) for such "standard" reasoning about local state abstraction.

By the definition in Figure 5, an expression is n-accepted by $IO^{std}\ P\ w$ if (1) it is n-accepted by $P\ w$ if it is already a value and (2) evaluating it to a value in $0 < i \leq n$ steps

$$IO^{std} : (W \rightarrow_{mon,ne} UPred(Val)) \rightarrow_{ne} (W \rightarrow_{ne} Pred(Cmd))$$

$$IO^{std}\ P\ w \overset{\mathsf{def}}{=}$$

$$\left\{ (n, cmd) \;\middle|\; \begin{array}{l} (n, cmd) \in P\ w \text{ if } cmd \in Val \text{ and,} \\ \text{for all } \sigma_r, \sigma_f, \sigma', 0 < i \le n, v. \sigma_r :_n w \wedge \\ \quad (\sigma_r \uplus \sigma_f, cmd) \to^i (\sigma', v) \Rightarrow \exists \sigma'_r, w' \sqsupseteq^{\text{pub}} w. \\ \quad\quad \sigma' = \sigma'_r \uplus \sigma_f \wedge \sigma'_r :_{n-i} w' \wedge (n-i, v) \in P\ w' \end{array} \right\}$$

$$\iota_l^{std} \overset{\mathsf{def}}{=} (l, =, =, H^{std})$$

$$H^{std}\ l\ w \overset{\mathsf{def}}{=} \left\{ (n, \{l \mapsto v\}) \;\middle|\; \begin{array}{l} n = 0 \text{ or } (n-1, v) \in \\ \quad \mathtt{JSVal}_{IO^{std}, Ref^{std}}\ (roll^{-1}\ w) \end{array} \right\}$$

$$Ref^{std} : W \rightarrow_{mon,ne} UPred(Loc)$$

$$Ref^{std}\ w \overset{\mathsf{def}}{=} \{(n, l) \mid \exists j.\ w(j) =_{n+1} \iota_l^{std}\}$$

Figure 5. An effect interpretation capturing standard local state abstraction.

in a store σ_r that is n-accepted by world w implies that the resulting store σ'_r is $n-i$-accepted by a public future world $w' \sqsupseteq^{\text{pub}} w$ and the resulting value is $n-i$ accepted by $P\ w''$. The original store is in fact allowed to additionally contain a *frame* part σ_f which must not be modified by the evaluation. This definition corresponds roughly to what one would typically find in the $\mathcal{E}$ relation of a Kripke logical relation, except that it only provides guarantees after reduction to a value, i.e. the evaluation is allowed to get stuck, and when it does, nothing is guaranteed about σ' and the resulting expression.[5]

We instantiate ρ to Ref^{std}, defined in Figure 5 w.r.t. an island ι_l^{std}. The island ι_l^{std} takes ownership of a location l and requires that its value satisfies $\mathtt{JSVal}_{IO^{std}, Ref^{std}}$. Ref^{std} defines as n-acceptable all locations l that are owned by such an island ι_l^{std}, or at least one that $n+1$-approximates it. The TR shows that effect interpretation (IO^{std}, Ref^{std}) is valid.

Instantiating our Fundamental Theorem with the effect interpretation (IO^{std}, Ref^{std}) produces a logical relation that captures the encapsulation of local state in λ_{JS} and can be used to reason about non-trivial capability patterns. The next sections demonstrate interesting examples of this: (1) a ticket dispenser function featuring a non-trivial protocol on private state, (2) the ad example from the introduction featuring a capability with restricted authority on shared data (the DOM) and (3) two isolated components with different authority on a shared LIFO communication channel.

4.1. Invariants and Protocols

Consider the following expression:

$$ticketDispenser \overset{\mathsf{def}}{=} \mathtt{func}(attacker)$$
$$\left\{ \begin{array}{l} \mathtt{let}(o = \mathtt{ref}\ 0) \\ \mathtt{let}(dispTkt = \mathtt{func}()\{\mathtt{let}(v = \mathtt{deref}\ o)\{o := v + 2; v\}\}) \\ attacker(dispTkt); \mathtt{deref}\ o \end{array} \right\}$$

5. This is appropriate for our untyped setting with stuck terms, but unusual as Kripke logical relations are most often used for static type systems that rule out stuck terms.

The expression takes an (untrusted) function argument $attacker$. The code allocates a new mutable reference o, initially 0 and constructs a function $dispTkt$. When called, the function increases o's value by 2 and returns the old value. If we assume for simplicity an infinite range of primitive integers, $dispTkt$ respects a protocol in its usage of o: its value will always remain even and will only ever *increase*. The attacker code is invoked and receives access to $dispTkt$. After the attacker code returns, the value of o is returned. Since the attacker has access to $dispTkt$ but not o and by inspecting $dispTkt$, we can expect the following to hold:

Lemma 1. *Take a store* σ, $\Sigma = \text{dom}(\sigma)$ *and a value* $\emptyset; \Sigma \vdash attacker$. *If* $(\sigma, ticketDispenser\ attacker) \to (\sigma', v)$, *then* v *is even and* ≥ 0.

With our Fundamental Theorem and effect interpretation (IO^{std}, Ref^{std}), we can prove that this holds.

Proof sketch. If the evaluation terminates, then for some $l \notin \Sigma$, it must factor as follows (omitting the body of $dispTkt$ for brevity) with $v = \sigma'(l)$:

$$\begin{array}{l} (\sigma, ticketDispenser\ attacker) \to^* \\ \quad (\sigma[l \mapsto 0], attacker\ (\mathtt{func}()\{\cdots\}); \mathtt{deref}\ l) \to^* \\ \quad\quad\quad (\sigma', (v'; \mathtt{deref}\ l)) \to^* (\sigma', \sigma'(l)) \end{array}$$

Define a world w with one island ι_l^{std} for every $l \in \Sigma$. Then for any n and $w' \sqsupseteq w$, we have $(n, w') \in [\![\Sigma]\!]_{IO^{std}, Ref^{std}}$.

The next island $\iota_{tkt,l,k}$ captures l's intended protocol:

$$\iota_{tkt,l,k} \overset{\mathsf{def}}{=} ((l,k), \sqsubseteq_{tkt}, \sqsubseteq_{tkt}, H_{tkt}) \qquad \text{for } k \text{ even}$$
$$(l,k) \sqsubseteq_{tkt} (l', k') \text{ iff } l = l' \wedge k' \geq k \wedge k', k \text{ even}$$
$$H_{tkt}\ (l,k)\ w \overset{\mathsf{def}}{=} \{(n, \{l \mapsto k\}) \mid n \in \mathbb{N}\}$$

Define $w' = w[j \mapsto \iota_{tkt,l,0}]$ for $j \notin \text{dom}(w)$. Clearly, $w' \sqsupseteq w$.

We show in the TR that for any n:

$$\begin{array}{l} (n, \mathtt{func}()\{\mathtt{return}\ (\mathtt{let}(v = \mathtt{deref}\ l)\{l := v + 2; v\})\}) \in \\ ([\mathtt{JSVal}_{IO^{std}, Ref^{std}}] \to IO^{std}\ \mathtt{JSVal}_{IO^{std}, Ref^{std}})\ w' \subseteq \\ \quad\quad\quad\quad \mathtt{JSVal}_{IO^{std}, Ref^{std}}\ w' \end{array}$$

The proof takes about two paragraphs. Essentially, it shows that executing the function in a store satisfying a world $w'' \sqsupseteq w'$ produces a new store satisfying a world $w''' \sqsupseteq^{\text{pub}} w''$. Specifically, the island $w''(j) = \iota_{tkt,l,k}$ will take a public transition to $w'''(j) = \iota_{tkt,l,k+2}$. The function's result value in such a store is a number, so satisfies $\mathtt{JSVal}_{IO^{std}, Ref^{std}}$.

By the Fundamental Theorem, $attacker$ n-satisfies $\mathcal{E}[IO^{std}\ \mathtt{JSVal}_{IO^{std}, Ref^{std}}]\ w'$. Then, $attacker\ (\mathtt{func}()\{\cdots\})$ must also n-satisfy $\mathcal{E}[IO^{std}\ \mathtt{JSVal}_{IO^{std}, Ref^{std}}]\ w'$ (by a standard lemma) and we can deduce that σ' must n-satisfy some $w'' \sqsupseteq^{\text{pub}} w'$, so that $\sigma'(l)$ must be even and ≥ 0. □

4.2. Restricted capabilities

Another important object capability pattern is to give a component *restricted* access to a resource, while other code keeps full authority over it. This can be implemented by giving the component access to a trusted object that has full access to the resource, but whose methods allow only restricted interaction with it. For reasoning about such a component with restricted authority, we use a world that carries two separate protocols for how the shared state can evolve. A first protocol dictates changes to the shared resource that the component itself has the authority to make while the second specifies changes that other code (with potentially greater authority) may make, and which the component under scrutiny should be able to deal with. These two protocols are given by, respectively, the public and private transitions in an island: a component is itself allowed to make public transitions, but must be able to cope with private transitions made by other code. In this section, we demonstrate this for the example from the introduction which restricts the authority of an untrusted advertisement in a client-side web page.

We repeat the example web page initialisation function:

$$\mathit{initWebPage} \overset{\text{def}}{=} \texttt{func}(\mathit{document}, \mathit{ad})$$
$$\left\{\begin{array}{l} \mathit{document.setProp}(\text{“}\mathit{someProperty}\text{”}, 42) \\ \texttt{let}\ (\mathit{adNode} = \mathit{document.addChild}(\text{“}\mathit{ad\_div}\text{”})) \\ \texttt{let}\ (\mathit{rAdNode} = \mathit{rnode}(\mathit{adNode}, 0)) \\ \mathit{ad.initialize}(\mathit{rAdNode}) \\ \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42 \end{array}\right\}$$

The idea is that $\mathit{initWebPage}$ receives access to a $\mathit{document}$ object that represents the entire web page and carries the authority to modify it. It uses a function rnode to construct a restricted capability $\mathit{rAdNode}$ for accessing and modifying only the ad's part of the page and passes that to the untrusted ad's initialisation code. We do not repeat the definition of rnode, which constructs an object that forwards method invocations to the underlying DOM node but returns `null` when asked for a node outside the ad's turf. If everything works correctly, we should be able to prove that if $\mathit{initWebPage}$ terminates, it must return `true`.

To formalise our assumptions about the behaviour of $\mathit{document}$ and its child nodes, we use a form of trees defined by the following grammar:

$$\mathit{tree} ::= v \mid (\mathit{id} \mapsto \mathit{tree})^* \qquad\qquad \mathit{id} \in \mathit{String}$$

We define a notion of plugging a subtree in a tree at a certain path (a list of ids) as follows:

$$t'[[] \mapsto t] \overset{\text{def}}{=} t$$
$$(\mathit{id}_1 \mapsto c_1, \cdots, \mathit{id}_n \mapsto c_n)[[p, \overline{p}] \mapsto t] \overset{\text{def}}{=}$$
$$\begin{cases} (\mathit{id}_1 \mapsto c_1 \cdots \mathit{id}_{j-1} \mapsto c_{j-1}, & \text{if } p = \mathit{id}_j \wedge \\ \quad \mathit{id}_j \mapsto t', \mathit{id}_{j+1} \mapsto c_{j+1} \cdots \mathit{id}_n \mapsto c_n) & \quad t' = c_j[\overline{p} \mapsto t] \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

We define an island $\iota^{\text{dom}}_{l,\mathit{tree},P}$ to govern the state of the DOM. It is parameterised by a function $P \in \mathit{String}^* \to W \to_{mon,ne} \mathit{UPred}(\mathit{Val})$, which defines, for every path in the DOM, a predicate that the DOM property at that path should satisfy.

$$\iota^{\text{dom}}_{l,\mathit{tree},P} \overset{\text{def}}{=} ((l, \mathit{tree}), \sqsubseteq^{\text{dom}}, \sqsubseteq^{\text{dom}}, H^{\text{dom}}_P)$$
$$(l', \mathit{tree}') \sqsupseteq^{\text{dom}} (l, \mathit{tree}) \text{ iff } l = l'$$

The store predicate H^{dom}_P is defined in the TR such that $H^{\text{dom}}_P\ (l, t)\ w$ accepts stores containing a representation of DOM tree t and for every property in the tree at path p, the value satisfies $P\ p\ w$. Note that the island's transition relation $\sqsupseteq^{\text{dom}}$ does not restrict the evolution of the tree.

For our ad example, we define a restricted transition relation $\sqsupseteq^{\text{r}-\text{dom}}_p$ expressing the restricted authority of the ad, i.e. only allowing changes to the DOM under path p:

$$(l_1, t_1) \sqsupseteq^{\text{r}-\text{dom}}_p (l_2, t_2) \text{ iff}$$
$$l_1 = l_2 \wedge (\forall t'_1, t_f.\ t_1 = t_f[p \mapsto t'_1] \Rightarrow \exists t'_2.\ t_2 = t_f[p \mapsto t'_2])$$

Lemma 2. *Assume that $\mathit{document}$'s methods $\mathit{getChild}$, parent, $\mathit{getProp}$, $\mathit{setProp}$, $\mathit{addChild}$ and $\mathit{delChild}$ behave in the "obvious" way (to be defined formally in the TR) in stores that contain the representation of a state of the DOM. If $w(j) = \iota^{\text{dom}}_{l,t,P}$ for some j, t and P, $\sigma :_n w$, and ad is a closed expression and $(\sigma, \mathit{initWebPage}(\mathit{document}, \mathit{ad})) \to^i (\sigma', v)$ for $i \leq n$, then $v = \texttt{true}$.*

The proof of this lemma is essentially based on the restricted transition relation $\sqsupseteq^{\text{r}-\text{dom}}$ mentioned above. However, it is complicated by the fact that DOM properties can be higher-order, i.e. functions that carry and use capabilities themselves. In the proof, we have to express that DOM properties under "$\mathit{ad\_div}$" have the same authority restriction as the ad. But what about the authority of trusted code (including DOM properties outside "$\mathit{ad\_div}$")? Naturally, this other code may modify the rest of the DOM, but can it modify the tree under "$\mathit{ad\_div}$"? Crucially, the code must not (by accident or malice) store capabilities with authority greater than the ad's (e.g. $\mathit{document}$ itself) inside the ad's reach. The most general solution is to require that they do not do this, i.e. that they preserve the authority bound of the ad. However, because $\mathit{initWebPage}$ terminates after the $\mathit{ad.initialize}$ call, and no trusted code gets a chance to run, this requirement is not in fact necessary for the lemma above. The proof in the TR relies on this simplification and does not restrict the DOM properties outside the ad's territory. However, we think it can be generalised if needed.

4.3. Isolated but communicating components

Another interesting pattern is similar to the previous example, but features *multiple* untrusted components that have restricted and *different* authority to a shared resource. In such a scenario, it is necessary to prevent the resource from being used as a communication channel for passing capabilities to the other component that it does not hold.

Consider a mashup page that embeds two disjoint pieces of untrusted code: $\mathit{attacker}_1$ and $\mathit{attacker}_2$. The mashup sets up a restricted communication channel: a stack that $\mathit{attacker}_1$ and $\mathit{attacker}_2$ can respectively push to and pop from. Figure 6 shows what the mashup's code could look like. If we suppose that $\mathit{attacker}_1$ and $\mathit{attacker}_2$ do not share a communication channel to begin with, then evaluating mashup in an arbitrary heap should always produce a non-negative number. Note that this example only works if the stack rejects non-constant values: if not, $\mathit{attacker}_1$ could push

```
mashup ≝ func(attacker1, attacker2)
{ let (stk = ref null)
  let (push = func(v)
      { if ¬isConstant(v) then undef else
          stk = ref({val = v, rest = deref stk}); undef })
  let (pop = func()
      { let (top = deref stk)
        if (top == null){undef}
          else {stk = (deref top).rest; (deref top).val}})
  let (size = func()
      { let (c = ref 0)
        let (top = ref (deref stk))
        while ((deref top) ! = null){
            c = deref c + 1; top = (deref top).rest}
        deref c })
  attacker1(push)
  let (s1 = size())
  attacker2(pop)
  s1 − size() }
```

Figure 6. A mashup application embedding two untrusted components, isolated from each other but with a restricted communication channel.

the *push* function itself on the stack, for *attacker*$_2$ to retrieve and use for stepping outside its pop-only authority.

Formally, the expected result is as follows:

Lemma 3. *If* $(n, attacker_i)$ *are in* $\texttt{JSVal}_{IO^{std},Ref^{std}}\ w_i$ *for* $i = 1, 2$ *and disjoint worlds* w_1 *and* w_2 *and if a store* σ *n-satisfies* $w_1 \oplus w_2$ *for* n *sufficiently large, then executing* $mashup(attacker_1, attacker_2)$ *in* σ*, produces a result* ≥ 0.

The premise that the $attacker_i$ are in $\texttt{JSVal}_{IO^{std},Ref^{std}}$ for disjoint worlds expresses the informal requirement that they do not share a communication channel to begin with. If they did, then *attacker1* could pass the *push* capability to *attacker2* and break our result. In practice, the requirement should be easily provable thanks to the absence of global mutable state in our capability-safe language. According to the Fundamental Theorem, the premise is satisfied, for example, if we know that the two pieces of code are well-formed in an empty context and empty store typing.

Proof sketch. In this proof sketch, all statements should be interpreted with respect to a sufficiently large step-index n.

We first define an island $\iota^{stack}_{(s,\bar{v})}$ capturing the stack's invariant; its states are the list of values in the stack and its private transition relation $\sqsupseteq^{stack}$ allows arbitrary modifications. We also define two restricted transition relations $\sqsupseteq^{stack\uparrow}$ and $\sqsupseteq^{stack\downarrow}$ that allow the stack to only grow or shrink respectively:

$$\iota^{stack}_{(s,\bar{v})} \stackrel{\mathsf{def}}{=} ((s,\bar{v}), \sqsupseteq^{stack}, \sqsupseteq^{stack}, H^{stack})$$

$$(l',\bar{v}') \sqsupseteq^{stack} (l,\bar{v}) \text{ iff } l = l'$$

$$(l',\bar{v}') \sqsupseteq^{stack\uparrow} (l,\bar{v}) \text{ iff } l = l' \wedge \exists \bar{v}''.\ \bar{v}' = \bar{v}''\bar{v}$$

$$(l',\bar{v}') \sqsupseteq^{stack\downarrow} (l,\bar{v}) \text{ iff } l = l' \wedge \exists \bar{v}''.\ \bar{v} = \bar{v}''\bar{v}'$$

$$H^{stack}\ (l, (v_1 \cdots v_n))\ w \stackrel{\mathsf{def}}{=}$$

$$\left\{ (n,\sigma) \;\middle|\; \begin{array}{c} \exists l_1, \cdots, l_n, l_{n+1}, v_1 \cdots v_n \in Const.\ \sigma(l) = l_1 \wedge \\ \mathrm{dom}(\sigma) = \{l, l_1, \cdots, l_n\} \wedge l_{n+1} = \texttt{null} \wedge \\ \forall i \in 1..n.\ \sigma(l_i) = \{val = v_i, rest = l_{i+1}\} \end{array} \right\}$$

After allocating a location l for *stk*, we define a world w_3, with $\mathrm{dom}(w_3) = \{j\}$, $w_3(j) = \iota^{stack}_{(l,\cdot)}$ for some j and the empty list $\cdot$. Next, we show that *push* is in $\texttt{JSVal}_{IO^{std},Ref^{std}}$ for world $w_3[j.\phi^{\mathrm{pub}} \mapsto \sqsupseteq^{stack\uparrow}]$ and *pop* in $\texttt{JSVal}_{IO^{std},Ref^{std}}$ for world $w_3[j.\phi^{\mathrm{pub}} \mapsto \sqsupseteq^{stack\downarrow}]$. We can derive that $attacker_1(push)$ is accepted by $\mathcal{E}[IO^{std}\ \texttt{JSVal}_{IO^{std},Ref^{std}}]$ in world $w_4 \stackrel{\mathsf{def}}{=} w_1 \oplus (w_3[j.\phi^{\mathrm{pub}} \mapsto \sqsupseteq^{stack\uparrow}])$ and we get that the resulting store is valid in a world $w_1' \oplus w_3'$ with $\mathrm{dom}(w_3') = \{j\}$ and $\mathrm{dom}(w_1') \supseteq \mathrm{dom}(w_1)$. From the fact that $w_1' \oplus w_3' \sqsupseteq^{\mathrm{pub}} w_4$, we know that the stack can only have grown. We can also derive that $attacker_2(pop)$ is accepted by $\mathcal{E}[IO^{std}\ \texttt{JSVal}_{IO^{std},Ref^{std}}]$ in a world $w_5 \stackrel{\mathsf{def}}{=} w_2 \oplus (w_3'[j.\phi^{\mathrm{pub}} \mapsto \sqsupseteq^{stack\downarrow}])$ and notice that the part of the store satisfying $H^{stack}(w_3'(j).s)\ (w_1' \oplus w_3')$ will also satisfy $H^{stack}(w_3'(j).s)\ w_5$ because (crucially) the definition of H^{stack} ignores its world argument, which it can because the content of the stack is restricted to first-order values. We then obtain a resulting world $w_2' \oplus w_3''$ with $\mathrm{dom}(w_3'') = \{j\}$ and $\mathrm{dom}(w_2') \supseteq \mathrm{dom}(w_2)$. We conclude from the fact that $w_2' \oplus w_3'' \sqsupseteq^{\mathrm{pub}} w_5$, that the stack can only have shrunk. Since *size* is called in a store satisfying $w_1' \oplus w_2' \oplus w_3''$, $s_1 - size()$ must give a non-negative result. □

5. Reference graph dynamics

The previous section shows how the effect interpretation (IO^{std}, Ref^{std}) can be used to verify results that rely on local state abstraction. However, it does not suffice for proving a set of reference graph properties that have previously been proposed as characteristic for object-capability languages. What is lacking is a way to restrict the primitive effects that an expression is allowed to produce. In this section, we introduce those properties, and in the next section, we discuss how they follow from effect parametricity.

The intended properties are standard in the object capability literature and have previously been formalised and proposed as a characterisation of capability-safety by Maffeis et al. [10]. For a programming language with a certain type of operational semantics, they characterise capability-safety through a formalisation of properties about the evolution of the reference graph. Here, we instantiate their formalism for λ_{JS} and explain that their properties are not sufficient to characterise capability-safety. Sometimes, Maffeis et al.'s definitions are more general than our instantiation of it, but not fundamentally stronger.

5.1. Capability safety as reference graph dynamics

We first introduce some notations used by Maffeis et al.: $e \sqsubseteq e'$ if e is a syntactic subterm of e'. Sets $\mathbb{D} \stackrel{\mathsf{def}}{=} \{\mathtt{r}, \mathtt{w}\}$ and $\mathbb{A} \stackrel{\mathsf{def}}{=} Loc \times \mathbb{D}$ represent read and write permissions and actions on references. For example, $(l, \mathtt{r})$ denotes reading location l. Allocating and initialising a new memory location is considered a combined reading ($\mathtt{r}$) and writing ($\mathtt{w}$) action. The *can influence*-relation on actions $(l,d) \triangleright (l',d')$ holds when $l = l'$, $d = \mathtt{w}$ and $d' = \mathtt{r}$. A set of actions $\mathcal{A}_1$ can influence another $\mathcal{A}_2$ ($\mathcal{A}_1 \triangleright \mathcal{A}_2$) when $a_1 \triangleright a_2$ for some $a_1 \in \mathcal{A}_1$, $a_2 \in \mathcal{A}_2$.

We define a labeled version of λ_{JS}'s impure evaluation judgement in Figure 7: $(\sigma, e) \rightarrow_A (\sigma, e)$ with $A \subseteq \mathbb{A}$. We further instantiate Maffeis et al.'s framework by defining $\mathrm{tCap}(e) \overset{\mathsf{def}}{=} \{l \mid l \sqsubseteq e\}$, i.e. the capabilities of an expression are the references that it syntactically contains, $\mathrm{priv}(l) \overset{\mathsf{def}}{=} \{\mathtt{r}, \mathtt{w}\}$ (i.e. a primitive reference provides reading and writing authority) and $\mathrm{cAuth}(\sigma, l)$ is the least set of actions A such that $\{(l,\mathtt{r}), (l,\mathtt{w})\} \subseteq A$ and $\left(\cup_{(l,\mathtt{r}) \in A} \mathrm{tCap}(\sigma(l)) \times \mathbb{D}\right) \subseteq A$. In other words, a reference l in store σ carries the authority to read and write values at location l and, recursively, the authority of those values.

Maffeis et al. characterise capability-safety by the following reference graph dynamics properties. For brevity, we define $\mathrm{nauth}(\sigma', \sigma) \overset{\mathsf{def}}{=} (\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)) \times \{\mathtt{r}, \mathtt{w}\}$. The map $\mathrm{auth}(\sigma, e) \overset{\mathsf{def}}{=} \bigcup_{c \in \mathrm{tCap}(e)} \mathrm{cAuth}(\sigma, c)$ must be a *valid authority map* for the language. This is by definition true iff $(\sigma, e) \rightarrow_A (\sigma', e')$ implies that

- RG-AUTH1: $A \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$
- RG-AUTH2: $\mathrm{auth}(\sigma', e') \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$

Additionally, if $(\sigma, e) \rightarrow_A^* (\sigma', v) \not\rightarrow$ and $v \in \mathit{Val}$, then for any location l, we must have:

- RG-CONN: $A \not\triangleright \mathrm{cAuth}(\sigma, l)$ implies that $\mathrm{cAuth}(\sigma', l) = \mathrm{cAuth}(\sigma, l) \cup \{(l,\mathtt{r}), (l,\mathtt{w})\}$
- RG-NOAMPL: $A \triangleright \mathrm{cAuth}(\sigma, l)$ implies that

$$\mathrm{cAuth}(\sigma', l) \subseteq \mathrm{cAuth}(\sigma, l) \cup \{(l,\mathtt{r}), (l,\mathtt{w})\} \cup \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$$

Property RG-CONN is known as "Only Connectivity Begets Connectivity" and RG-NOAMPL as "No Authority Amplification".

5.2. Reference graph dynamics are not enough

The above properties are *necessary but not sufficient* to characterise object-capability languages. They constitute an *over-approximation* of the authority of a term, known as the *topology-only bound on authority*. The approximation is imprecise because it considers indirect references equivalent to direct references and as such ignores objects' behaviour.

Consider, for example, the ticket dispenser example from Section 4.1, where attacker code was given access to a function $\mathit{dispTkt} = \mathtt{func}()\{\mathtt{let}(v = \mathtt{deref}\ o)\{o := v + 2; v\}\}$ but no direct access to reference o. The topology-only bound does not distinguish an expression with a reference to $\mathit{dispTkt}$ or a direct reference to o, so the safety of our ticket dispenser cannot follow. In fact, none of the results from Section 4 can be proven using just the topology-only bound on authority.

To be clear, the problem is not just that the soundness of those examples is *hard* to prove using the topology-only bound. Rather, their soundness does not follow because the bound is not strong enough. One can prove this by constructing a language that satisfies the bound but invalidates the examples, for example by adding a $\mathrm{deepInspect}$ primitive that returns the set of references held by an arbitrary value, i.e. $\mathrm{deepInspect}(\mathit{dispTkt})$ evaluates to the singleton list $[o]$. More details about this argument are deferred to the TR.

6. Reference Graph Dynamics from Effect Parametricity

Although the reference graph dynamics properties from the previous section are not sufficient to reason about examples like those in Section 4 or to characterise capability safety, they may still be of interest in some applications. In this section, we explain how our Fundamental Theorem implies results that are analogous but a bit more semantic.

First, we need to explain that we cannot derive the properties themselves because of the more semantic nature of our logical relations. Consider, for example, an expression e and a location l that e does not mention (i.e. $l \not\sqsubseteq e$). Now take x fresh and consider $e' \overset{\mathsf{def}}{=} \mathtt{let}(x = l)\ e$. Syntactically, e' holds a reference to capability l and Maffeis et al. would consider it to have greater authority, despite the semantic fact that e' never uses l. However, e' purely evaluates to e, so they are equivalent for our logical relations and any results we prove will not distinguish them. Nevertheless, we can prove properties that are analogous but more semantic.

6.1. An Effect Interpretation for Memory Regions

We start from an effect interpretation that formulates a region memory discipline, defined in Figure 8. It uses a special-purpose island $\iota^{\mathrm{rgn}}_{j,L}$ to define the current set of addresses in a memory region. This set may grow over time (as expressed by the island's transition relation), with newly allocated references or with existing references whose ownership is passed to the region. j is the index of the island in the world. We require that regions are isolated from one another and the rest of the world, i.e. values stored in region j must themselves never access memory outside the region. Formally, the heap invariant H^{rgn}_j enforces this by requiring that values in the region are accepted by $\mathtt{JSVal}_{IO^{rgn}_j, Ref^{rgn}_j}$.

We instantiate ρ as Ref^{rgn}_j, accepting only region j's current addresses, and μ as IO^{rgn}_j, which essentially accepts expressions that only access memory within region j and otherwise respect the world invariants and authority bounds. More formally, $IO^{rgn}_j\ P\ w\ n$-accepts expressions e that are n-accepted by P if they are values and such that executing them in i steps in a w-acceptable store σ_r will only access memory locations inside region j. The resulting expression must $n - i$-satisfy $\mathcal{E}[IO^{rgn}_j\ P]$ in a publicly accessible extension w' of world w. The resulting world must extend the region with all freshly allocated references. Additionally, the store is allowed to contain an additional *frame* part σ_f not governed by w which must be left unmodified.

Contrary to IO^{std}, the definition of IO^{rgn}_j does not assume that the evaluation $(\sigma_r \uplus \sigma_f, \mathit{cmd}) \rightarrow^i_A (\sigma', e')$ produces an e' that is a value. This means that IO^{rgn}_j provides guarantees about arbitrary computations, not just those that terminate successfully. Such definitions are typically only

$$\frac{e_1 \hookrightarrow e_2}{(\sigma, E\langle e_1\rangle) \rightarrow_{\emptyset} (\sigma, E\langle e_2\rangle)}\ (\textsc{E-Pure}) \qquad \frac{l \notin \mathrm{dom}(\sigma) \quad A = \{(l,\mathbf{r}),(l,\mathbf{w})\}}{(\sigma, E\langle \texttt{ref}\ v\rangle) \rightarrow_A (\sigma[l \mapsto v], E\langle l\rangle)}\ (\textsc{E-Ref})$$

$$(\sigma, E\langle \texttt{deref}\ l\rangle) \rightarrow_{\{(l,\mathbf{r})\}} (\sigma, E\langle \sigma(l)\rangle) \qquad (\textsc{E-Deref})$$
$$(\sigma, E\langle l = v\rangle) \rightarrow_{\{(l,\mathbf{w})\}} (\sigma[l \mapsto v], E\langle v\rangle) \qquad (\textsc{E-SetRef})$$

Figure 7. Action-labeled version $\rightarrow_A$ of the impure λ_{JS} evaluation judgement. Multi-step versions $\rightarrow^i_A$ and $\rightarrow^*_A$ accumulate labels of substeps.

$$\iota^{rgn}_{j,L} \stackrel{\mathsf{def}}{=} (L, \subseteq, \subseteq, H^{rgn}_j)$$
$$\mathit{Ref}^{rgn}_j : W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Loc})$$
$$\mathit{Ref}^{rgn}_j\ w \stackrel{\mathsf{def}}{=} \left\{(n,l) \;\middle|\; w(j) = (L,\subseteq,\subseteq,H), l \in L \wedge H =_{n+1} H^{rgn}_j\right\}$$
$$\mathit{IO}^{rgn}_j : (W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Val})) \rightarrow W \rightarrow_{ne} \mathit{Pred}(\mathit{Cmd})$$
$$\mathit{IO}^{rgn}_j\ P\ w \stackrel{\mathsf{def}}{=}$$
$$\left\{\begin{array}{l} (n, \mathit{cmd}) | \\ \quad (\mathit{cmd} \in \mathit{Val} \Rightarrow (n,\mathit{cmd}) \in P\ w) \wedge \\ \quad \forall L, H.\ \text{if } w(j) = (L,\subseteq,\subseteq,H) \wedge H =_{n+1} H^{rgn}_j \text{ then} \\ \quad \forall 0 < i \leq n, \sigma_r :_n w, \sigma_f.\ (\sigma_r \uplus \sigma_f, \mathit{cmd}) \rightarrow^i_A (\sigma', e') \Rightarrow \\ \quad \exists \sigma'_r, w' \sqsupseteq^{\mathrm{pub}} w,\ \text{for } L' = L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)). \\ \quad w'(j) = (L',\subseteq,\subseteq,H) \wedge \sigma' = \sigma'_r \uplus \sigma_f \wedge A \subseteq (L' \times \mathbb{D}) \wedge \\ \qquad\qquad (n-i, e') \in \mathcal{E}[\mathit{IO}^{rgn}_j\ P]\ w' \wedge \sigma'_r :_{n-i} w' \end{array}\right\}$$
$$H^{rgn}_j : \mathcal{P}(\mathit{Loc}) \rightarrow \hat{W} \rightarrow_{mon,ne} \mathit{UPred}(\mathrm{Store})$$
$$H^{rgn}_j\ L\ w \stackrel{\mathsf{def}}{=}$$
$$\left\{(n,\sigma) \;\middle|\; \begin{array}{l} \mathrm{dom}(\sigma) = L \text{ and for all } l \in L.\ n = 0 \text{ or} \\ \quad (n-1, \sigma(l)) \in \texttt{JSVal}_{\mathit{IO}^{rgn}_j, \mathit{Ref}^{rgn}_j}\ (\mathit{roll}^{-1}\ w) \end{array}\right\}$$

Figure 8. Region-based effect interpretation for memory access bounds.

found in logical relations for *concurrent* programming languages, but in our case we want memory access bounds even for evaluations that do not terminate or end up in a stuck state. Formally, it does complicate the definition of IO^{rgn} because if e' is potentially not a value, it makes no sense to require that it is accepted by $P\ w'$. The solution is to require recursively that it is accepted by $\mathcal{E}[\mathit{IO}^{rgn}_j\ P]$.

6.2. Memory access bounds

With this effect interpretation, our Fundamental Theorem implies memory access bounds similar to the reference graph dynamics properties. To emphasise the correspondence, we define $\texttt{memBound}(\sigma, e, L)$ as a judgement analogous to (but more semantic than) the statement $\mathrm{auth}(\sigma, e) = L \times \mathbb{D}$.

Definition 1. *We define* $\texttt{memBound}(\sigma, e, L)$ *for a store* σ *and* $L \subseteq \mathrm{dom}(\sigma)$ *iff for* $j = 1$, $w = [j \mapsto \iota^{rgn}_{j,L}]$ *and any* n*, we have:*

- $(n, e) \in \mathcal{E}[\mathit{IO}^{rgn}_j\ \texttt{JSVal}_{\mathit{IO}^{rgn}_j, \mathit{Ref}^{rgn}_j}]\ w$
- $\exists \sigma_f, \sigma_r. \sigma = \sigma_r \uplus \sigma_f$ *and* $\sigma_r :_n w$

The Fundamental Theorem implies that $\texttt{memBound}(\sigma, e, L)$ holds whenever $\mathrm{auth}(\sigma, e) = L \times \mathbb{D}$ (proof in TR):

Lemma 4. *If* $\mathrm{auth}(\sigma, e) = L \times \mathbb{D}$*, then* $\texttt{memBound}(\sigma, e, L)$.

Furthermore, the memBound judgement satisfies properties akin to the reference graph dynamics properties. The following property corresponds to properties RG-AUTH1 and RG-AUTH2 in the previous section. It specifies that memBound bounds an expression's memory access and that the property is preserved by evaluation.

Lemma 5. *If* $\texttt{memBound}(\sigma_1, e_1, L)$ *and* $(\sigma_1, e_1) \rightarrow^i_A (\sigma_2, e_2)$*, then for* $L' = L \cup (\mathrm{dom}(\sigma_2) \setminus \mathrm{dom}(\sigma_1))$

- $A \subseteq L' \times \mathbb{D}$ *and* $\sigma_2|_{\mathrm{dom}(\sigma_1)\setminus L} = \sigma_1|_{\mathrm{dom}(\sigma_1)\setminus L}$.
- $\texttt{memBound}(\sigma_2, e_2, L')$.

The next two properties correspond to RG-CONN and RG-NOAMPL. The first states that evaluating an expression whose authority *cannot influence* another expression's, leaves that other expression's memBound unaffected. The latter specifies that evaluating an expression whose authority *can influence* another expression's, may increase that other expression's memBound, but not beyond the union of the two expressions' original bounds and newly allocated locations.

Lemma 6. *If* $(\sigma, e) \rightarrow^*_A (\sigma', v)$*,* $A \not\triangleright L \times \mathbb{D}$ *and* $\texttt{memBound}(\sigma, e', L)$*, then still* $\texttt{memBound}(\sigma', e', L)$.

Lemma 7. *If* $\texttt{memBound}(\sigma, e_1, L_1)$*,* $\texttt{memBound}(\sigma, e_2, L_2)$*,* $(\sigma, e_1) \rightarrow^*_A (\sigma', v)$ *and* $A \triangleright L_2 \times \mathbb{D}$*, then we have* $\texttt{memBound}(\sigma', e_2, L')$ *for* $L' = L_1 \cup L_2 \cup (\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma))$.

7. Related Work

Object capability research has a long history, possibly starting with Dennis and Van Horn's proposed hardware protection primitives [2]. Later work on operating systems and processor hardware with capability security primitives is surveyed by Levy [21]. More recent work includes Capsicum (primitive capabilities offered by FreeBSD kernel primitives) [4] and CHERI (processor-level object capabilities combined with virtual memory) [5]. A thorough overview of the object capability model and the capability-safe programming language E is in Miller's PhD thesis [3]. Other capability-safe languages include Joe-E [6], Emily [8], W7 [22], Newspeak [9] and Google Caja [7].

Two papers formalise reference graph properties for capability systems using an operational semantics. The first is Maffeis et al.'s paper, presented in Section 5. The second is Shapiro and Weber's verification of the confinement properties of EROS [23]. They prove a property related to one of the properties in Section 5, which also ignores the behaviour of processes/objects.

Dimoulas et al. formalise capability safety in terms of a notion of component boundaries and the ownership of code by security principals [24], in a simple language without mutable state. The formalisation of capability-safety does not seem intended for reasoning about code. Instead, they propose to extend the object capability language with alternative security mechanisms, that enable specific types of reasoning: built-in dynamic access control and an information flow type system. Similarly, Drossopoulou and Noble argue to extend object capability languages with a kind of declarative policies [25]. Generally, we are not convinced that modular enforcement of typical policies requires additional security mechanisms in the language. Rather, they can be implemented cleanly and modularly using standard patterns. Proving that such implementations enforce a declarative policy can be done using techniques as developed here.

Spiessens studied the safety of capability patterns using Knowledge Behaviour Models and a logic called SCOLL [11], [12]. His goal of validating the safety of capability patterns is the same as ours, but he reasons at a higher level of abstraction, viewing executable code as interacting abstract entities with a behaviour specification. As a result, his results do not directly apply to concrete code in a concrete language, but both the implementations and the language must be separately verified to satisfy their specification. Spiessens' automatic approximation of future behaviour imposes some restrictions in the logic, like the absence of non-monotonic authority changes.

Taly et al. automatically analyse security-critical APIs in a secure subset of JavaScript to guarantee that API implementations do not leak references to objects marked as internal [26]. A limitation is that they only deal with leaking of direct references. Establishing security requires separate verification of objects *not* marked as internal.

Garg et al. [27] and Jia et al. [28] have proposed and studied powerful program logics for *interface-confined* code in, respectively, a first-order and higher-order setting. Like us, they prove properties of arbitrary, untyped attacker code and can reason about memory as well as other effects. They do this for *interface-confined* code, a syntactic requirement that appears similar to the restrictions in an object-capability language. However, the work does not intend to cover object-capability languages, does not look into previous notions of capability-safety (see Section 5), and the authors state that they cannot model object-capability languages in general. Specifically, there is no direct way to reason about untrusted code that gets access to some primitive pointers, but not all, although this can be modeled using additional indirection. It is not clear to what extent the work supports patterns with separate authority over shared data structures as discussed in Section 4.

A second category of related work is on logical relations for proving encapsulation properties in higher-order languages. The logical relations we use are (unary) step-indexed Kripke logical relations [13], [29]. We have generally followed the notations used by Birkedal et al. [16] and we used their recipe for defining recursive worlds using ultrametric space theory. Our worlds are inspired by Dreyer et al.'s [17]. The region-based effect interpretation used in Section 6 produces a logical relation related to one used by Thamsborg et al. for proving relational results about a region-based type-and-effect system [18].

There is a long line of work on using logical relations for proving local state abstraction results [17], [30]–[35]. The work covers increasingly complex languages (e.g. stack variables vs. first-order heap vs. higher-order heap) and uses increasingly complex Kripke worlds for specifying invariants and protocols on the evolution of acceptable/related stores. Recursive types and higher-order heaps are dealt with using either step-indexing like us or more complex (but perhaps more elegant) solutions based on domain theory.

Our work is closely related to this line of work, but there are some differentiating aspects. First, we work for an untyped language. Although it is known that step-indexing enables logical relations for untyped languages [36], we are (to our knowledge) the first to demonstrate this for a language with a mutable store. Our Kripke worlds are based on and similar to those of Dreyer et al. [17], but some points are novel. Our use of public/private transitions to model restricted authority over a shared resource, and the fact that the public transition relation in our islands can grow are both (to the best of our knowledge) novel. The idea enables a kind of rely-guarantee reasoning, as demonstrated in Section 4.3 and the approach bears a resemblance to permission assertions on a shared region, as in, for example, Dinsdale-Young et al.'s *Concurrent Abstract Predicates* [37]. Also novel is our notion of effect interpretations as a way to define custom world-indexed restrictions on primitive effects, as well as the axioms prescribing compatibility of the effect interpretation to the publicly accessible effects of the language. Finally, perhaps most importantly, the link to object capabilities and our characterisation of capability-safety are novel.

8. Conclusion

In summary, this paper presents a novel approach for formal reasoning about a capability-safe programming language. We use state-of-the-art techniques from programming language research for capturing the encapsulation of the language, supplemented with some additional ideas for capturing specificities of the model. Our demonstration of three typical but non-trivial patterns in Section 4 and our derivation of reference graph dynamics results in Section 6 shows that our approach is significantly more powerful than previous approaches and sufficiently powerful for realistic examples. We have presented our technique for a subset of λ_{JS}, a relatively simple core calculus for JavaScript, but we expect that it generalises to other settings including assembly languages with primitive capabilities [5], [38], capability-safe subsets of JavaScript and typed capability-safe languages [6], [8], [9]. In the TR, we discuss how our techniques generalise to *relational* rather than unary properties.

Our work also offers new insight into the nature of capability-safe programming languages. Summarily, the property groups three characteristics, all captured by our model: (a) encapsulation of local state: a quite common feature in programming languages like Java, ML, JavaScript etc. (b) absence of global mutable state: a less common feature which is nevertheless crucial for isolating components from each other (like the example in Section 4.3) and, finally, (c) primitive effects only available through primitive capabilities, so authority to produce primitive effects can be controlled by giving components access to the capability or not.

With the better understanding of capability-safety, this work creates the potential for a program logic (perhaps with automated tool support) that can be used to conveniently reason about code in a capability-safe language. Specifically, our Fundamental Theorem tells us what semantic properties such a logic or tool can soundly offer as axioms over untrusted code. As such, this paper contributes a key semantic understanding, but the design of a program logic or automated tool remains future work.

Acknowledgements

Dominique Devriese holds a postdoctoral mandate from the Research Foundation Flanders (FWO). This research is partially funded by project grants from the Research Fund KU Leuven, and from the Research Foundation Flanders (FWO). This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

[1] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *European Symposium on Security and Privacy*. IEEE, 2016.

[2] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[3] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[4] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *USENIX Security*. USENIX, 2010.

[5] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ISCA*. IEEE, 2014, pp. 457–468.

[6] A. Mettler, D. Wagner, and T. Close, "Joe-E: A security-oriented subset of Java," in *NDSS*, 2010.

[7] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized JavaScript," Google, Tech. Rep., 2008.

[8] M. Stiegler, "Emily: A high performance language for enabling secure cooperation," in *C5*. IEEE, 2007, pp. 163–169.

[9] G. Bracha, P. von der Ah, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda, "Modules as objects in Newspeak," in *ECOOP*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6183, pp. 405–428.

[10] S. Maffeis, J. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *S&P*. IEEE, 2010, pp. 125–140.

[11] F. Spiessens, "Patterns of safe collaboration," Ph.D. dissertation, Université Catholique de Louvain, 2007.

[12] F. Spiessens and P. Van Roy, "A practical formal model for safety analysis in capability-based systems," in *TGC*, ser. LNCS. Springer Berlin Heidelberg, 2005, vol. 3705, pp. 248–278.

[13] A. W. Appel and D. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 5, pp. 657–683, Sep. 2001.

[14] A. Ahmed, A. Appel, and R. Virga, "A stratified semantics of general references embeddable in higher-order logic," in *LICS*. IEEE, 2002, pp. 75–86.

[15] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *ECOOP*. Springer, 2010, pp. 126–150.

[16] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang, "Step-indexed Kripke models over recursive worlds," in *POPL*. ACM, 2011, pp. 119–132.

[17] D. Dreyer, G. Neis, and L. Birkedal, "The impact of higher-order state and control effects on local relational reasoning," *J. Funct. Program.*, vol. 22, no. 4–5, pp. 477–528, 2012.

[18] J. Thamsborg and L. Birkedal, "A Kripke logical relation for effect-based program transformations," in *ICFP*. ACM, 2011, pp. 445–456.

[19] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, 1991.

[20] P. Wadler, "Comprehending monads," *Math. Struct. Comp. Sci.*, vol. 2, pp. 461–493, 12 1992.

[21] H. M. Levy, *Capability-based computer systems*. Digital Press Bedford, 1984, vol. 12.

[22] J. A. Rees, "A security kernel based on the lambda-calculus," Ph.D. dissertation, MIT, 1995.

[23] J. Shapiro and S. Weber, "Verifying the EROS confinement mechanism," in *S&P*. IEEE, 2000, pp. 166–176.

[24] C. Dimoulas, S. Moore, A. Askarov, and S. Chong, "Declarative policies for capability control," in *CSF*. IEEE, 2014, pp. 3–17.

[25] S. Drossopoulou and J. Noble, "The need for capability policies," in *FTfJP*. ACM, 2013, pp. 6:1–6:7.

[26] A. Taly, Ù. Erlingsson, J. Mitchell, M. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs," in *S&P*. IEEE, May 2011, pp. 363–378.

[27] D. Garg, J. Franklin, D. Kaynar, and A. Datta, "Compositional system security with interface-confined adversaries," in *MFPS*, ser. ENTCS. Elsevier, 2010, vol. 265, pp. 49 – 71.

[28] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *CSF*. IEEE, July 2015, pp. 512–525.

[29] A. J. Ahmed, "Semantics of types for mutable state," Ph.D. dissertation, Princeton University, 2004.

[30] P. W. O'Hearn and R. D. Tennent, "Parametricity and local variables," *J. ACM*, vol. 42, no. 3, pp. 658–709, 1995.

[31] U. Reddy and H. Yang, "Correctness of data representations involving heap data structures," *Sci. Comput. Program.*, vol. 50, no. 1/3, p. 129/160, 2004.

[32] A. Banerjee and D. A. Naumann, "Ownership confinement ensures representation independence for object-oriented programs," *J. ACM*, vol. 52, no. 6, pp. 894–960, Nov. 2005.

[33] N. Bohr and L. Birkedal, "Relational reasoning for recursive types and references," in *PLaS*, ser. LNCS. Springer Berlin Heidelberg, 2006, vol. 4279, pp. 79–96.

[34] L. Birkedal, K. Støvring, and J. Thamsborg, "Relational parametricity for references and recursive types," in *TLDI*. ACM, 2009, pp. 91–104.

[35] A. Ahmed, D. Dreyer, and A. Rossberg, "State-dependent representation independence," in *POPL*. ACM, 2009, pp. 340–353.

[36] A. M. Pitts, "Step-indexed biorthogonality: a tutorial example," in *Modelling, Controlling and Reasoning About State*, ser. Dagstuhl Seminar Proceedings. LZfI, 2010, no. 10351.

[37] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *ECOOP*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6183, pp. 504–528.

[38] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ASPLOS*. ACM, 1994, pp. 319–327.

This appendix contains two things: a supplementary, less mature section about the generalisation of our results to relational properties and proofs and details about the results in the paper. For clarity, we repeat theorems and lemmas from the paper (with the same numbering) and then additionally provide proofs.

Appendix A. From Predicates to Relations

A.1. Binary effect parametricity

In a binary setting, we use uniform step-indexed *relations*, over sets A: $URel(A) \stackrel{\text{def}}{=} UPred(A \times A)$. For a relation $P \in URel(A)$, we say that $a_1, a_2 \in A$ are n-related by P if $(n, a_1, a_2) \in P$. The definition of our Kripke worlds changes only slightly in that the heap predicate $H \in \text{State} \to \text{StorePred}$ of an island becomes a heap relation $H \in \text{State} \to \text{StoreRel}$ with

$$\text{StoreRel} \stackrel{\text{def}}{=} \{\psi \in \hat{W} \to_{mon,ne} URel(\text{Store})\}$$

Rather than defining when a store is acceptable in the current state of the island, the heap relation now defines when two stores are related in the current state of the island. For example, it may relate stores when they contain (potentially different) representations of the same abstract data structure.

The judgement $\sigma :_n w$ is generalised to a judgement $(\sigma_1, \sigma_2) :_n w$

$$\begin{aligned}(\sigma_1, \sigma_2) :_n w \text{ iff } & \exists \sigma_{1,1} \cdots \sigma_{m,1}, \sigma_{1,2} \cdots \sigma_{m,2}.\ \sigma_k = \sigma_{1,k} \uplus \cdots \uplus \sigma_{m,k} \text{ for } k = 1..2 \\ & \text{and } \forall j \in \text{dom}(w).\ (n, \sigma_{j,1}, \sigma_{j,2}) \in w(j).H(w(j).s)\ (roll\ w)\end{aligned}$$

In order to construct a binary version of our $\texttt{JSVal}_{\mu,\rho}$ predicate of acceptable values, we first need binary versions of our *Cnst*, $\{P\}$, $(P \cup P')$, $\mathcal{E}[P]$ and $[P] \to P'$ predicates. The most interesting of these is $\mathcal{E}[P]$, because it is interestingly asymmetric.

$$\begin{aligned}&\mathcal{E}[P''] : W \to_{ne} Rel(Expr) \\ &\mathcal{E}[P'']\ w \stackrel{\text{def}}{=} \left\{ (n, e_1, e_2) \;\middle|\; \begin{array}{l} \text{for all } i \le n, e_1'.\ (e_1 \to^i e_1' \text{ and} \\ \quad e_1' \in Cmd) \Rightarrow \exists e_2' \in Cmd.\ e_2 \to^* e_2' \wedge (n-i, e_1', e_2') \in P''\ w \end{array} \right\}\end{aligned}$$

For a command relation P'', the relation $\mathcal{E}[P'']$ n-relates an expression e_1 to e_2 if evaluating it for $i \le n$ steps to a command e_1' implies that e_2 must also evaluate to a command e_2' such that e_1' is $n-i$-related to e_2' by P''. This definition is asymmetric: we only learn something if e_1' evaluates to a command. One sometimes speaks of logical approximation rather than logical similarity to express this asymmetricity. These asymmetric definitions are often used, because they tend to work much better than symmetric ones (if the latter work at all). One can always define symmetric relations later as approximation in two directions.

$$\begin{aligned}
Cnst &: W \to_{mon,ne} URel(Val) \\
Cnst\ w &\stackrel{\text{def}}{=} \{(n, c, c) \mid n \in \mathbb{N}, c \in Const\} \\
\{P\} &: W \to_{mon,ne} URel(Val) \\
\{P\}\ w &\stackrel{\text{def}}{=} \left\{(n, \{\overline{str : v_1}\}, \{\overline{str : v_2}\}) \mid \text{for all } i < n, \overline{(i, v_1, v_2) \in P\ w}\right\} \\
P \cup P' &: W \to_{mon,ne} URel(Val) \\
(P \cup P')\ w &\stackrel{\text{def}}{=} P\ w \cup P'\ w \\
([P] \to P'') &: W \to_{mon,ne} URel(Val) \\
([P] \to P'')\ w &\stackrel{\text{def}}{=} \{(n, \texttt{func}(x_1 \cdots x_k)\{\texttt{return } e_1\}, \texttt{func}(x_1 \cdots x_k)\{\texttt{return } e_2\}) \mid \\
&\qquad \text{for all } v_{1,1} \cdots v_{k,1}, v_{1,2} \cdots v_{k,2}, w' \sqsupseteq w, i < n.\ \overline{(i, v_{j,1}, v_{j,2}) \in P\ w'} \Rightarrow \\
&\qquad\qquad (i, e_1[x_1/v_{1,1}, \cdots, x_n/v_{k,1}], e_2[x_1/v_{1,2}, \cdots, x_n/v_{k,2}]) \in \mathcal{E}[P'']\ w' \}
\end{aligned}$$

The relation *Cnst* relates constants only to itself for arbitrary n. For value relations P and P', the relations $\{P\}$ and $P \cup P'$ relate record literals with related fields and values that are related by either P or P'. For a command predicate P'', $[P] \to P''$ relates functions that map P-related argument values to P''-related results in future worlds and step-indices $i < n$.

Effect interpretations become pairs (μ, ρ) with $\rho : W \to_{mon,ne} URel(Loc)$ and

$$\mu : (W \to_{mon,ne} URel(Val)) \to_{ne} (W \to_{ne} Rel(Cmd)).$$

The binary version of $\texttt{JSVal}_{\mu,\rho}$ is then not much different:

$$\texttt{JSVal}_{\mu,\rho} : W \to_{mon,ne} URel(Val)$$

$$\texttt{JSVal}_{\mu,\rho} \overset{\text{def}}{=} Cnst \cup \rho \cup \{\texttt{JSVal}_{\mu,\rho}\} \cup ([\texttt{JSVal}_{\mu,\rho}] \to \mu\ \texttt{JSVal}_{\mu,\rho})$$

The axioms for a *valid* effect interpretation also generalise easily:

- A-PURE: If $(n, v_1, v_2) \in P\ w$ then $(n, v_1, v_2) \in \mu\ P\ w$.
- A-BIND: If $(n, e_1, e_2) \in \mu\ P\ w$ and $(n', E_1\langle v_1\rangle, E_2\langle v_2\rangle) \in \mathcal{E}[\mu\ P']\ w'$ for all $n' \leq n$, $w' \sqsupseteq w$ and $(n', v_1, v_2) \in P\ w'$, then $(n, E_1\langle e_1\rangle, E_2\langle e_2\rangle) \in \mathcal{E}[\mu\ P']\ w$.
- A-ASSIGN: If $(n, e_{1,1}, e_{1,2}) \in \texttt{JSVal}_{\mu,\rho}\ w$ and $(n, e_{2,1}, e_{2,2}) \in \texttt{JSVal}_{\mu,\rho}\ w$, then $(n, e_{1,1} = e_{2,1}, e_{1,2} = e_{2,2}) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$.
- A-DEREF: If $(n, e_1, e_2) \in \texttt{JSVal}_{\mu,\rho}\ w$, then $(n, \texttt{deref}\ e_1, \texttt{deref}\ e_2) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$.
- A-REF: If $(n, e_1, e_2) \in \texttt{JSVal}_{\mu,\rho}\ w$, then $(n, \texttt{ref}\ e_1, \texttt{ref}\ e_2) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$.

Finally, the Fundamental Theorem now relates any well-formed term to itself. First, we generalise the semantic interpretations of the reference and value contexts Σ and Γ of the well-formedness judgement:

$$[\![\Sigma]\!]_{\mu,\rho} : UPred(W)$$

$$[\![\Sigma]\!]_{\mu,\rho} \overset{\text{def}}{=} \{(n, w) \mid \text{for all } l \in \Sigma.(n, l, l) \in \rho\ w\}$$

$$[\![\Gamma]\!]_{\mu,\rho}\ w : URel(Val^{\Gamma})$$

$$[\![\Gamma]\!]_{\mu,\rho}\ w \overset{\text{def}}{=} \{(n, \gamma_1, \gamma_2) \mid \forall x \in \Gamma.(n, \gamma_1(x), \gamma_2(x)) \in \texttt{JSVal}_{\mu,\rho}\ w\}$$

And we can now formulate a binary version of our Fundamental Theorem:

Theorem 4 (Fundamental Theorem, Binary Version)**.** *If* $\Gamma, \Sigma \vdash e$ *then for a valid effect interpretation* (μ, ρ) *and for all* n, γ_1, γ_2 *and* w *with* $(n, w) \in [\![\Sigma]\!]_{\mu,\rho}$ *and* $(n, \gamma_1, \gamma_2) \in [\![\Gamma]\!]_{\mu,\rho}\ w$, $(n, \gamma_1(e), \gamma_2(e))$ *must be in* $\mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$.

A.2. Relational Applications

Examples of interesting relational properties are not hard to think of. Consider, for example, again the example of a web page embedding an untrusted ad. The idea of that example was to give the ad access to an object $rnode(adNode, nil)$ where $adNode$ is the DOM node representing the ad's reserved space on the web page. The function $rnode$ was constructed so that this did not allow the ad to access the rest of the web page, and in Section 4.2, we have shown that, effectively, the ad can indeed not modify the web page outside $adNode$.

However, we have not proven other security-relevant correctness properties of $rnode$. Specifically, we also want to be sure that the ad cannot *read* the contents of the web page outside of its reserved space. Formally, this can be expressed using a non-interference property: differences in the web page outside of the ad's reserved space may not influence its behaviour. Additionally, we would also like to be sure that the $rnode$ function does not hinder well-behaving ads, i.e. ads that do not try to escape from their sandbox should behave the same with access to the restricted object as with direct access to the original.

Both of these properties are naturally relational. Without making everything fully formal for space concerns, let us sketch, for example, how we might prove the former property about the ad not reading outside its space. Recall the abstract trees that we used to abstractly specify a state of the DOM. Assume that we have two such trees t_1 and t_2, two paths p_1 and p_2 and an abstract tree t representing the state of the ad's turf. Then we can construct an appropriate island $\iota_{t_1,p_1,t_2,p_2,t}$ that considers two stores related if one contains a representation of tree $t_1[p_1 \mapsto t]$ and the other $t_2[p_2 \mapsto t]$. Then, if $adNode_1$ and $adNode_2$ are DOM nodes representing the node at path p_i in DOM t_i, then we can show that $rnode(adNode_1, nil)$ and $rnode(adNode_2, nil)$ are related at a binary generalisation of $\texttt{JSVal}_{IO^{std},Ref^{std}}$ in any world containing island $\iota_{t_1,p_1,t_2,p_2,t}$, to express that they will behave identically. Applying the binary Fundamental Theorem to the untrusted ad's code will also tell us that it is related to itself in $\texttt{JSVal}_{IO^{std},Ref^{std}}$. From this, we can show that $ad.initialize(rnode(adNode_i, 0))$ for $i = 1..2$ will evaluate to related values in stores representing the different DOMs and leave the stores in a related state.

$$IO^{std} : (W \to_{mon,ne} URel(Val)) \to_{ne} (W \to_{ne} Rel(Cmd))$$

$$IO^{std}\ P\ w \overset{\text{def}}{=}$$

$$\left\{ (n, e_1, e_2) \;\middle|\; \begin{array}{l} e_2 \in Val \wedge (n, e_1, e_2) \in P\ w \text{ if } e_1 \in Val \text{ and,} \\ \text{for all } \sigma_{r,1}, \sigma_{r,2}, \sigma_{f,1}, \sigma_{f,2}, \sigma'_1, 0 < i \leq n, e'. \\ ((\sigma_{r,1}, \sigma_{r,2}) :_n w,\ (\sigma_{r,1} \uplus \sigma_{f,1}, e_1) \to^i (\sigma'_1, e'_1) \text{ and } e'_1 \in Val) \Rightarrow \\ \quad \exists \sigma'_{r,1}, \sigma'_{r,2}, w' \sqsupseteq^{\text{pub}} w, e'_2 \in Val.\ \sigma'_1 = \sigma'_{r,1} \uplus \sigma_{f,1} \text{ and} \\ \qquad (\sigma_{r,2} \uplus \sigma_{f,2}, e_2) \to^* (\sigma'_{r,2} \uplus \sigma_{f,2}, e'_2) \wedge \\ \qquad\quad (\sigma'_{r,1}, \sigma'_{r,2}) :_{n-i} w' \text{ and } (n - i, e'_1, e'_2) \in P\ w' \end{array} \right\}$$

A.3. Relational Effect Interpretations

Finally, we want to mention the role that effect interpretations can play in a binary setting. In the unary setting, they allow us to require/enforce restrictions on the use of primitive effects, such as accessing only one region of memory (see Section 5). In the binary setting, this generalises to requiring/enforcing relations between different interpretations of primitive effects.

This is especially powerful when code accesses these primitive effects in a way that can be overridden, as is often the case in capability-safe programming languages. Consider, for example, a version of λ_{JS} extended with console output, invoked through the print method of a primitive object `out` that is not publicly accessible, but only provided as an argument to the main method.

$$v ::= \cdots \mid \texttt{out}$$

$$\frac{}{(\sigma, E\langle \texttt{out}.\mathit{print}(\mathit{str})\langle\rangle \rightarrow_{\mathit{str}} (\sigma, E\langle \texttt{undef}\rangle)}$$

Suppose now that we want to give a piece of code access to a simulated output channel that simply keeps a log of all output. Because actual output is generated through the `out` object's print method, we can simply construct an object with the same interface as `out` to achieve this. First, we construct an object that offers a print method as well as a method getLog to inspect the current log.

$$\mathit{loggingOut} \overset{\mathsf{def}}{=} \texttt{func}()\left\{\begin{array}{l}\texttt{let}\ \ \mathit{log} = \texttt{ref}\ \mathit{nil}\ \texttt{in} \\ \left\{\begin{array}{l}\mathit{getLog} = \texttt{func}()\{\texttt{deref}\ \ \mathit{log}\} \\ \mathit{print} = \texttt{func}(v)\ \{\mathit{log} = \mathit{cons}(v, \texttt{deref}\ \ \mathit{log})\}\end{array}\right\}\end{array}\right\}$$

Now consider what happens if we provide an expression e with a reference to an object with only such an alternative print function:

$$\begin{array}{l} e' \overset{\mathsf{def}}{=} \texttt{let}\ \ \mathit{log} = \mathit{loggingOut}() \\ \qquad e(\{\mathit{print} = \mathit{log}.\mathit{print}\}); \\ \qquad \mathit{log}.\mathit{getLog}() \end{array}$$

We then expect that providing such a reference instead of `out` itself to e achieves our goal of logging all output, but does not otherwise modify the execution. In other words, e' should evaluate as $(\sigma, e') \rightarrow^* (\sigma'', \mathit{strs})$ for some σ'' if $(\sigma, e(\texttt{out})) \rightarrow^*_{\mathit{strs}} (\sigma', v)$ for some v.

For proving this property, we could proceed by constructing an island governing the state of the output log:

$$\begin{array}{r} \iota^{\mathit{log}}_{l,\bar{str}} \overset{\mathsf{def}}{=} ((l, \bar{str}), \sqsupseteq^{\mathit{log}}, \sqsupseteq^{\mathit{log}}, H^{\mathit{log}}) \\ (l, \bar{str}') \sqsupseteq (l', \bar{str}) \text{ iff } l = l' \wedge \bar{str} \text{ is prefix of } \bar{str}' \\ H^{\mathit{log}}\ \ (l, \bar{str})\ \ w \overset{\mathsf{def}}{=} \{(n, \emptyset, \{l \mapsto \bar{str}\})\} \end{array}$$

The island keeps a location l and a list of strings that represent the current location and content of the log. Note how the heap relation relates an empty store on the left (i.e. the log is not stored in memory) but only on the right.

We could now use the following effect interpretation:

$$IO^{\mathit{log}}_j\ \ P\ w \overset{\mathsf{def}}{=}$$

$$\left\{ (n, e_1, e_2) \;\middle|\; \begin{array}{l} (n, e_1, e_2) \in P \text{ if } e_1 \in \mathit{Val} \wedge \\ \text{for all } \sigma_1, \sigma_2, \sigma'_1, i \leq n, e'_1, \overline{\mathit{str}}. \\ \big(w(j) = \iota^{\mathit{log}}_{l,\overline{\mathit{str}}} \wedge (\sigma_1, \sigma_2) :_n w \wedge (\sigma_1, e_1) \rightarrow^{i}_{\overline{\mathit{str}'}} (\sigma'_1, e'_1) \wedge e'_1 \in \mathit{Val}\big) \Rightarrow \\ \ \exists \sigma'_2, w' \sqsupseteq^{\mathrm{pub}} w. (\sigma'_1, \sigma'_2) :_{n-i} w' \wedge \\ \qquad (\sigma_2, e_2) \rightarrow^*_{\emptyset} (\sigma'_2, e'_2) \wedge (n - i, e'_1, e'_2) \in P\ \ w' \wedge \\ \qquad\quad w'(j) = \iota^{\mathit{log}}_{l,(\mathit{rev}(\overline{\mathit{str}'}) +\!\!+ \overline{\mathit{str}})} \end{array} \right\}$$

In other words, IO^{log}_j n-relates e_1 and e_2 if termination of e_1 with output $\overline{\mathit{str}'}$ implies termination of e_2 with *no* output, related result values and stores, and an accurate update of the log.

If we now prove that, together with $\mathit{Ref}^{\mathit{std}}$, this constitutes a valid effect interpretation, prove that `out` and $\{\mathit{print} = \mathit{log}.\mathit{print}\}$ are n-related by $\texttt{JSVal}_{IO^{\mathit{log}}_j, \mathit{Ref}^{\mathit{std}}}$, we can prove the desired property.

This example is interesting because it demonstrates how our effect interpretations allows specifying custom restrictions/relations on primitive effects, rather than just on the state of the store.

The remainder of this appendix provides details and proofs for the results in the paper.

Appendix B. Proofs and details for Section 2

Lemma 8. *For all v, $v \not\to$ and $v \not\to_o$.*

Proof. Simple case analysis. □

Lemma 9. *If $E\langle e\rangle$ is a value, then $E = \cdot$ and e is a value.*

Proof. Easy case analysis on E. □

Lemma 10. *If $cmd = E\langle e\rangle$, then e is also a command.*

Proof. We know that a cmd is either a value or $E_0\langle cmd_0\rangle$ for some $cmd_0 = v.\texttt{print}\,(v')$. If cmd is a value, then $E = \cdot$ and $e = cmd$ is a command. If $cmd = E_0\langle cmd_0\rangle$, then we prove by induction on E that e is a command.

1) $E = \cdot$: Then $e = cmd$ is a command.
2) $E = \texttt{let}(x = E')\ e'$. Because $cmd = E_0\langle cmd_0\rangle$ and $cmd = E\langle e\rangle = \texttt{let}(x = E'\langle e\rangle)\ e'$, there must be a E_0' such that $E_0 = \texttt{let}(x = E_0')\ e'$. This means that $E'\langle e\rangle = E_0'\langle cmd_0\rangle$ is also a command. By induction, we have that e must be a command.
3) $E = \texttt{if}(E')\{e\}\,\texttt{else}\,\{e\}$. Similar to case 2.
4) $E = E'; e$. Similar to case 2.
5) $E = E'.\texttt{print}\,(e')$. Because $cmd = E_0\langle cmd_0\rangle$ and $cmd = E\langle e\rangle = (E'\langle e\rangle).\texttt{print}\,(e')$, there must either be an E_0' such that $E_0 = E_0'.\texttt{print}\,(e')$. In which case, we continue as before. Or, $E_0 = \cdot$ and $cmd_0 = e.\texttt{print}\,(e')$ and e and e' are values. But values are commands, so we're good.
6) $E = v.\texttt{print}\,(E')$. Similar to case 5.

□

Lemma 11. *For all cmd, $cmd \not\to$.*

Proof. Suppose that $cmd \to e'$. By definition, this means that $cmd = E\langle e_1\rangle$, $e' = E\langle e_2\rangle$ and $e_1 \hookrightarrow e_2$. By Lemma 10, this means that e_1 is a command. But case analysis on $e_1 \hookrightarrow e_2$ tells us that this is not possible. □

Lemma 12. *If $E\langle e\rangle = E'\langle e'\rangle$, then one of the following holds:*

- *e is a value.*
- *e' is a value.*
- *$E = E'\langle E''\rangle$ with $E'' \neq \cdot$.*
- *$E' = E\langle E''\rangle$.*

Proof. Simple induction on the syntax of E. □

Lemma 13. *If $E\langle e\rangle \to e'$, then either e is a value or $e' = E\langle e''\rangle$ for some e'' and $e \to e''$.*

Proof. $E\langle e\rangle \to e'$ implies by definition that $E\langle e\rangle = E'\langle e_1\rangle$, $e' = E'\langle e_2\rangle$ and $e_1 \hookrightarrow e_2$. By Lemma 12, this implies that e is a value (in which case we're done), or e_1 is a value (but then $e_1 \hookrightarrow e_2$ is impossible) or one of the two following cases holds:

- $E = E'\langle E''\rangle$ with $E'' \neq \cdot$. But then $E'\langle e_1\rangle = E\langle e\rangle = E'\langle E''\langle e\rangle\rangle$, so that $e_1 = E''\langle e\rangle$. A simple case analysis on $e_1 \hookrightarrow e_2$ and E'' then implies that e must be a value, so we're done.
- $E' = E\langle E''\rangle$. Then $E\langle e\rangle = E'\langle e_1\rangle = E\langle E''\langle e_1\rangle\rangle$, so that $e = E''\langle e_1\rangle$. Then $e' = E'\langle e_2\rangle = E\langle E''\langle e_2\rangle\rangle$, and $e = E''\langle e_1\rangle \to E''\langle e_2\rangle$, so we're done.

□

Lemma 14. *If $E\langle e\rangle \to_o e'$, then either e is a value or $e' = E\langle e''\rangle$ for some e'' and $e \to_o e''$.*

Proof. We know that the evaluation must either be pure, in which case Lemma 13 tells us what we need, or $E\langle e\rangle = E'\langle\texttt{out}.\texttt{print}\,(str)\rangle$, $o = [str]$ and $e' = E'\langle\texttt{undef}\rangle$. By Lemma 12, then either e is a value (in which case we're done), or $\texttt{out}.\texttt{print}\,(str)$ is a value (but this is not possible) or one of the following cases holds:

- $E = E'\langle E''\rangle$ with $E'' \neq \cdot$. This means that $E\langle e\rangle = E'\langle E''\langle e\rangle\rangle = E'\langle\texttt{out}.\texttt{print}\,(str)\rangle$, so that $E''\langle e\rangle = \texttt{out}.\texttt{print}\,(str)$. This implies that $e = \texttt{out}$ or $e = str$, so in both cases, e is a value.

- $E' = E\langle E''\rangle$. In this case, $E\langle e\rangle = E'\langle \texttt{out.print}\ (str)\rangle = E\langle E''\langle \texttt{out.print}\ (str)\rangle\rangle$, so that $e = E''\langle \texttt{out.print}\ (str)\rangle$. Also, $e' = E'\langle \texttt{undef}\rangle = E\langle E''\langle \texttt{undef}\rangle\rangle$ and $e = E''\langle \texttt{out.print}\ (str)\rangle \to_o E''\langle \texttt{undef}\rangle$.

□

Lemma 15. *$\mathcal{E}[P] \supseteq P$ for all P.*

Proof. Take $cmd \in P$ and we prove that $cmd \in \mathcal{E}[P]$. Suppose that $cmd \to^* cmd'$. Then by Lemma 11, the evaluation must be trivial and $cmd' = cmd$. Then clearly, $cmd \in P$. □

Lemma 16. *If $e \to^* e'$, then $e \in \mathcal{E}[P]$ iff $e' \in \mathcal{E}[P]$.*

Proof. Suppose $e \to^* e'$ and $e' \in \mathcal{E}[P]$. Suppose that $e \to^* cmd$. Because evaluation is deterministic, it's easy to prove that $e' \to^* cmd$, so that $cmd \in P$.

Conversely, suppose $e \to^* e'$ and $e \in \mathcal{E}[P]$. Suppose $e' \to^* cmd$. Then we can concatenate the evaluations to get $e \to^* cmd$, which implies that $cmd \in P$. □

Theorem 1 (Fundamental Theorem for $\lambda_{\texttt{out},FO}$)**.** *For a valid effect interpretation (μ, ρ), $\Gamma \vdash e$ implies that for any $\gamma \in [\![\Gamma]\!]_{\mu,\rho}$, $\gamma(e)$ is in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.*

Proof. By induction on the $\Gamma \vdash e$ well-scopedness judgement. We have not named the rules in Figure 2, but it should be clear which ones we mean. By Lemma 15, it is sufficient (though not necessary) to prove that $\gamma(e) \in \mu\ \ Val_{\mu,\rho}$ and because $Val_{\mu,\rho}$ contains only values, Axiom A-PURE implies that it is sufficient (though again not necessary) to prove that $\gamma(e) \in Val_{\mu,\rho}$.

1) $\Gamma \vdash x$ if $x \in \Gamma$. Because $\gamma \in [\![\Gamma]\!]_{\mu,\rho}$, we know by definition that $\gamma(x) \in Val_{\mu,\rho}$.
2) $\Gamma \vdash v$ with $v \neq \texttt{out}$. By definition, this implies $v \in PureVal \subseteq Val_{\mu,\rho}$ and we know that $\gamma(v) = v$.
3) $\Gamma \vdash \texttt{let}(x = e_1)\ e_2$ with $\Gamma \vdash e_1$ and $\Gamma, x \vdash e_2$. We have that $\gamma(\texttt{let}(x = e_1)\ e_2) = \texttt{let}(x = \gamma(e_1))\ \gamma(e_2)$. We will show that this expression is in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $\texttt{let}(x = \gamma(e_1))\ \gamma(e_2) \to^* cmd$. Take the largest prefix of this judgement such that $\texttt{let}(x = \gamma(e_1))\ \gamma(e_2) \to^* \texttt{let}(x = e_1')\ \gamma(e_2)$.
If the evaluation ends there, then $\texttt{let}(x = e_1')\ \gamma(e_2) = cmd$ is a command and by Lemma 10, so is e_1'. If the evaluation does not end after reducing $\gamma(e_1)$ to a command, then e_1' must be a value by Lemma 13, so also a command.
The induction hypothesis for e_1 and the sub-evaluation $\gamma(e_1) \to^* e_1'$ now gives us that $e_1' \in \mu\ \ Val_{\mu,\rho}$. We also have that $\texttt{let}(x = e_1')\ \gamma(e_2) = (\texttt{let}(x = \cdot)\ \gamma(e_2))\langle e_1'\rangle$. Furthermore, if $v \in Val_{\mu,\rho}$ then $\texttt{let}(x = v)\ \gamma(e_2) \to \gamma'(e_2)$ with $\gamma' = \gamma[x \mapsto v]$. It easily follows that $\gamma' \in [\![\Gamma, x]\!]_{\mu,\rho}$ and by the induction hypothesis, we have that $\gamma'(e_2) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$. By Lemma 16, this implies that $\texttt{let}(x = v)\ \gamma(e_2) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$. We can now apply Axiom A-BIND to obtain that $\texttt{let}(x = e_1')\ \gamma(e_2) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
Now take the further evaluation (if any) $\texttt{let}(x = e_1')\ \gamma(e_2) \to^* cmd$. It follows directly that $cmd \in \mu\ \ Val_{\mu,\rho}$ as required.
4) $\Gamma \vdash \texttt{if}(e_1)\{e_2\}\,\texttt{else}\,\{e_3\}$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$, $\Gamma \vdash e_3$. Similar to case 3.
5) $\Gamma \vdash e_1; e_2$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. Similar to case 3.
6) $\Gamma \vdash \texttt{while}(e_1)\{e_2\}$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. We have that $\gamma(\texttt{while}(e_1)\{e_2\}) = \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}$. We will show that this expression is in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $\texttt{while}(\gamma(e_1))\{\gamma(e_2)\} \to^* cmd$. This evaluation must factor as follows:
$$\texttt{while}\,(\gamma(e_1))\{\gamma(e_2)\} \to$$
$$\texttt{if}\,(e_1)\{e_2; \texttt{while}(e_1)\{e_2\}\}\,\texttt{else}\,\{\texttt{undef}\}$$
$$\to^* cmd.$$
Now take the largest prefix of this evaluation such that it is of the form
$$\texttt{if}\,(e_1)\{e_2; \texttt{while}(e_1)\{e_2\}\}\,\texttt{else}\,\{\texttt{undef}\} \to^*$$
$$\texttt{if}\,(e_1)\{e_2; \texttt{while}(e_1)\{e_2\}\}\,\texttt{else}\,\{\texttt{undef}\}.$$
The rest of this case proceeds like case 3.
7) $\Gamma \vdash e_1.\texttt{print}\,(e_2)$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. We have that $\gamma(e_1.\texttt{print}\,(e_2)) = \gamma(e_1).\texttt{print}\,(\gamma(e_2))$. We will show that this expression is in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $\gamma(e_1).\,\texttt{print}\,(\gamma(e_2)) \to^* cmd$. Take the longest prefix such that $\gamma(e_1).\,\texttt{print}\,(\gamma(e_2)) \to^* e_1'.\,\texttt{print}\,(\gamma(e_2))$. If the evaluation stops there, then e_1' has to be a command because of Lemma 10. If not, e_1' must be a value, so also a command. The sub-evaluation $\gamma(e_1) \to^* e_1'$ together with the induction hypothesis then give us that $e_1' \in \mu\ \ Val_{\mu,\rho}$. By Lemma 16 and Axiom A-BIND, it now suffices to prove that for $v \in Val_{\mu,\rho}$, $v.\texttt{print}(\gamma(e_2)) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$. So, assume that $v.\texttt{print}\,(\gamma(e_2)) \to^* cmd$. Take the longest prefix such that $v.\texttt{print}\,(\gamma(e_2)) \to^* v.\texttt{print}\,(e_2')$. If the evaluation stops there, then e_2' has to be a command because of Lemma 10. If not, e_2' must be a value, so also a command. The sub-evaluation $\gamma(e_2) \to^* e_2'$ together with the induction hypothesis then give us that $e_2' \in \mu\ \ Val_{\mu,\rho}$. By Lemma 16 and Axiom A-BIND, it now suffices to prove that for $v' \in Val_{\mu,\rho}$, $v.\,\texttt{print}\,(v') \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
By Lemma 15, it suffices to prove that $v.\,\texttt{print}\,(v') \in \mu\ \ Val_{\mu,\rho}$. Axiom A-PRINT tells us immediately that this is true.

□

Lemma 17. *If* $e \rightarrow_{o}^{*} cmd$ *then the evaluation factors as* $e \rightarrow^{*} e' \rightarrow_{o}^{*} cmd$ *with* $e' \in Cmd$.

Proof. Easy induction on the length of the evaluation and the derivation of the impure evaluation. □

Lemma 18. *If* $e \in \mathcal{E}[IO_{triv}\ P]$ *and* $e \rightarrow_{o}^{*} v$*, then* $o = []$ *and* $v \in P$.

Proof. By Lemma 17, we can factor the evaluation as $e \rightarrow^{*} e' \rightarrow_{o}^{*} v$ with $e' \in Cmd$. We then easily get that $e' \in IO_{triv}\ P$ and from that $o = []$ and $v \in P$. □

The following Lemma was used in the proof of Example 1.

Lemma 19. (IO_{triv}, Ref_{triv}) *is a valid effect interpretation.*

Proof. We prove that the four axioms hold:

- A-RHO-OUT: $Ref_{triv} \subseteq \{\texttt{out}\}$. Direct.
- A-PURE: For a value $v \in P$, v must also be in $IO_{triv}\ P$. Assume that $v \rightarrow_{o}^{*} v'$. Lemma 8 tells us that the evaluation is trivial, $v = v'$ and $o = []$.
- A-BIND: If $e \in IO_{triv}\ P$ and $E\langle v\rangle \in \mathcal{E}[IO_{triv}\ P']$ for all values $v \in P$, then $E\langle e\rangle \in \mathcal{E}[IO_{triv}\ P']$. Assume that $E\langle e\rangle \rightarrow^{*} cmd$. We know that e is a command, so by Lemmas 13 and 11, either the evaluation is trivial (zero steps, $cmd = E\langle e\rangle$) or e is a value. In the latter case, our assumption about E directly gives us the result we need. In the former case, $E\langle e\rangle = cmd$ and we prove that it is in $IO_{triv}\ P'$. So, assume that $E\langle e\rangle \rightarrow_{o}^{*} v$. Take the longest prefix such that the evaluation is of the form $E\langle e\rangle \rightarrow_{o'}^{*} E\langle e'\rangle$. If the evaluation ends there, then $E\langle e'\rangle$ is a value, so by Lemma 9, e' is a value. If the evaluation continues, then by Lemma 14, e' must also be a value.
  The sub-evaluation $e \rightarrow_{o'}^{*} e'$ now gives us that $o' = []$ and $e' \in P$. It follows that $E\langle e'\rangle \in \mathcal{E}[IO_{triv}\ P']$. The remaining execution then gives us by Lemma 18 that $o = []$ and $v \in P'$.
- A-PRINT: If $v \in Val_{IO_{triv}, Ref_{triv}}$ then $v.\texttt{print}\ (v') \in IO_{triv}\ Val_{IO_{triv}, Ref_{triv}}$. Assume that $v.\texttt{print}\ (v') \rightarrow_{o}^{*} v$. It follows directly that the evaluation must be a single evaluation step, derived using evaluation rule E-OUT. However, this implies that $v = \texttt{out}$, but $Val_{IO_{triv}, Ref_{triv}}$ does not contain $\texttt{out}$, so this is a contradiction.

□

Now, we move to the extended calculus $\lambda_{\texttt{out},HO}$.

From here on, we will need some theory about metric spaces. We refer to Birkedal et al. for a short review of metric space theory [16, §2.1].

Lemma 20. $UPred(A)$ *is a complete bounded ultrametric space when the distance function is defined as follows. For* $P \in UPred(A)$*, define* $P_{[k]} \overset{\text{def}}{=} \{(m, cmd) \in P \mid m < k\}$*. Then, for* $P, Q \in UPred(A)$*, define the following metric (i.e. distance function between two predicates):*

$$d(P,Q) \overset{\text{def}}{=} \begin{cases} 2^{-\max\{k \mid P_{[k]} = Q_{[k]}\}} & \text{if } P \neq Q \\ 0 & \text{otherwise} \end{cases}$$

Proof. This lemma is already mentioned in Birkedal et al. [16]. It can be proven using standard techniques from metric space theory. □

Note that the lemma above does not depend on any properties of the set A, but works for arbitrary sets.

We will use some simple lemmas about $UPred(X)$:

Lemma 21. *For* $P, Q \in UPred(X)$*, we have that* $d(P,Q) \leq 2^{-n}$ *iff* $P_{[n]} = Q_{[n]}$.

Proof. If $P = Q$, there is nothing to prove. If not, then we call $m = \max\{k \mid P_{[k]} = Q_{[k]}\}$ and observe that $d(P,Q) = 2^{-m}$. We have that $d(P,Q) \leq 2^{-n}$ iff $m \geq n$ and $m \geq n$ iff $P_{[n]} = Q_{[n]}$. □

Note that in Section 2, we have left it implicit that the function $\mu \in UPred(Val) \rightarrow UPred(Cmd)$ should be an arrow in the category of complete bounded ultrametric spaces $\mathbf{CBUlt}_{ne}$, i.e. a non-expansive function.

Lemma 22. $d(\mathcal{E}[P], \mathcal{E}[Q]) \leq d(P,Q)$.

Proof. Because $UPred(X)$ is bisected, it suffices to prove that if $d(P,Q) \leq 2^{-n}$ then $d(\mathcal{E}[P], \mathcal{E}[Q]) \leq 2^{-n}$, so by Lemma 39, it suffices to prove that if $P_{[n]} = Q_{[n]}$, then $\mathcal{E}[P]_{[n]} = \mathcal{E}[Q]_{[n]}$ or by symmetry $\mathcal{E}[P]_{[n]} \subseteq \mathcal{E}[Q]_{[n]}$. So, take $(k, e) \in \mathcal{E}[P]_{[n]}$, i.e. $k < n$. We will show that also $(k, e) \in \mathcal{E}[Q]_{[n]}$, or (sufficiently) that $(k, e) \in \mathcal{E}[Q]$. So, take $i \leq k$ and $e \rightarrow^{i} cmd$. Then $(k - i, cmd) \in P$ and because $k - i \leq k < n$, also $(k - i, cmd) \in Q$. □

Lemma 23. $\mathcal{E}[P]$*,* $P_1 \rightarrow P_2$ *and* $Val_{\mu,\rho}$ *are uniform and well-defined if* P*,* P_1*,* P_2 *are uniform and* (μ, ρ) *is a valid effect interpretation.*

Proof. We need to prove a number of things:

- $\mathcal{E}[P]$ is uniform if P is uniform. Take $(n, e) \in \mathcal{E}[P]$ and $k \leq n$. Then we prove that (k, e) is also in $\mathcal{E}[P]$. Take $i \leq k$ and assume that $e \to^i cmd$. Then $i \leq k \leq n$, so $(n-i, cmd) \in P$ and by uniformity of P and $n - i \geq k - i$ also $(k-i, cmd) \in P$.
- $P_1 \to P_2$ are uniform if P_1 and P_2 are uniform. For $(n, \lambda x.e) \in P_1 \to P_2$ and $k \leq n$, we prove that $(k, \lambda x.e) \in P_1 \to P_2$. So, take $i \leq k$ and $(i, v) \in P_1$. Then obviously $i \leq k \leq n$ and $(i, e[x/v]) \in \mathcal{E}[P_2]$.
- We prove that $Val_{\mu,\rho}$ is well-defined (i.e. a solution to its defining equation exists and is unique) using the standard Banach fixpoint lemma. This means that we need to prove that the functor which maps a complete bounded predicate P to the predicate $f(P) = (\mathbb{N} \times PureVal) \cup \rho \cup (P \to \mu\ P)$ actually produces a uniform predicate and is contractive. The first is quite clear because $(\mathbb{N} \times PureVal)$ is easily shown to be uniform, ρ is assumed to be and so is $\mu\ P$ when P is uniform and $P \to \mu\ P$ when P and $\mu\ P$ are uniform. Finally, it's easy to see that taking the union of uniform predicates produces a uniform predicate.
  For proving that f is contractive, take $P, Q \in UPred(Val)$, we need to prove that $d(f(P), f(q)) \leq \frac{1}{2} d(P, Q)$. If P and Q are equal, then so are $f(P)$ and $f(Q)$ and there is nothing to prove. If not, it suffices to prove that if $P_{[k]} = Q_{[k]}$ then $(f(P))_{[k+1]} = (f(Q))_{[k+1]}$ or by symmetry $(f(P))_{[k+1]} \subseteq (f(Q))_{[k+1]}$. So, take $(n, e) \in (f(P))_{[k+1]}$. Note that this implies that $n < k+1$, i.e. $n \leq k$. By definition of f, we have that either $(n, e) \in (\mathbb{N} \times PureVal)$ or $(n, e) \in \rho$, in which case also $(n, e) \in f(Q)$, or $(n, e) \in P \to \mu\ P$. This implies that $e = \lambda x.e'$ for some e'. We will prove that then also $(n, e) \in Q \to \mu\ Q$. So, take $i < n$ and $(i, v) \in Q$, then it suffices to prove that $(i, e'[x/v]) \in \mathcal{E}[\mu\ Q]$. Because $i < n \leq k$ and $P_{[k]} = Q_{[k]}$, we have that (i, v) is also in P and so $(i, e'[x/v])$ must be in $\mathcal{E}[\mu\ P]$, because $(n, e) \in P \to \mu\ P$. By Lemma 39, we know that $d(P, Q) \leq 2^{-k}$. Because μ is non-expansive, we know that $d(\mu\ P, \mu\ Q) \leq 2^{-k}$ and by Lemma 22 also $d(\mathcal{E}[\mu\ P], \mathcal{E}[\mu\ Q]) \leq 2^{-k}$. Again, by Lemma 39 and the fact that $i < n \leq k$, this means that $(i, e'[x/v])$ is also in $\mathcal{E}[\mu\ Q]$, concluding our proof that f is contractive.

□

Lemma 24. $P \subseteq \mathcal{E}[P]$.

Proof. Take $(k, e) \in P$, $i \leq k$ and $e \to^i cmd$. Because of Lemma 11, i must be zero and $e = cmd$. It is then clear that $(k - i, cmd) = (k, e) \in P$. □

Lemma 25. *If* $e \to^i e'$*, then* $(n, e) \in \mathcal{E}[P]$ *iff* $(n-i, e') \in \mathcal{E}[P]$.

Proof. Suppose $e \to^i e'$ and $(n - i, e') \in \mathcal{E}[P]$. Suppose that $i' \leq n$ and $e \to^{i'} cmd$. Because evaluation is deterministic, it's easy to prove that $e' \to^{i'-i} cmd$, so that $(n - i - (i' - i), cmd) = (n - i', cmd) \in P$.

Conversely, suppose $e \to^i e'$ and $(n, e) \in \mathcal{E}[P]$. Suppose $i' \leq n - i$ and $e' \to^{i'} cmd$. Then we can concatenate the evaluations to get $e \to^{i+i'} cmd$, which implies that $(n - i - i', cmd) \in P$. □

Theorem 2 (Fundamental Theorem for $\lambda_{\mathtt{out},HO}$)**.** *For a valid effect interpretation* (μ, ρ)*,* $\Gamma \vdash e$ *implies for any* n *and* $(n, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}$ *that* $(n, \gamma(e))$ *is in* $\mathcal{E}[\mu\ Val_{\mu,\rho}]$.

Proof. By induction on the judgement $\Gamma \vdash e$. As before, we have not named the rules in Figure 2 and the additional one for lambdas, but it should be clear which ones we mean. By Lemma 24, it is sufficient (though not necessary) to prove that $(n, \gamma(e)) \in \mu\ Val_{\mu,\rho}$ and because $Val_{\mu,\rho}$ contains only values, Axiom A-PURE implies that it is sufficient (though again not necessary) to prove that $(n, \gamma(e)) \in Val_{\mu,\rho}$.

- $\Gamma \vdash x$ if $x \in \Gamma$. Because $(n, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}$, we know by definition that $(n, \gamma(x)) \in Val_{\mu,\rho}$.
- $\Gamma \vdash v$ with $v \neq \mathtt{out}$. By definition, this implies $(n, v) \in (\mathbb{N} \times PureVal) \subseteq Val_{\mu,\rho}$ and we know that $\gamma(v) = v$.
- $\Gamma \vdash \mathtt{let}(x = e_1)\ e_2$ with $\Gamma \vdash e_1$ and $\Gamma, x \vdash e_2$. We have that $\gamma(\mathtt{let}(x = e_1)\ e_2) = \mathtt{let}(x = \gamma(e_1))\ \gamma(e_2)$. We will show that this expression is n-acceptable in $\mathcal{E}[\mu\ Val_{\mu,\rho}]$, so assume that $i \leq n$ and $\mathtt{let}(x = \gamma(e_1))\ \gamma(e_2) \to^i cmd$. Take the largest prefix of this judgement such that $\mathtt{let}(x = \gamma(e_1))\ \gamma(e_2) \to^{i'} \mathtt{let}(x = e_1')\ \gamma(e_2)$.
  If the evaluation ends there, then $\mathtt{let}(x = e_1')\ \gamma(e_2) = cmd$ is a command and by Lemma 10, so is e_1'. If the evaluation does not end after reducing $\gamma(e_1)$ to e_1', then e_1' must be a value by Lemma 13, so also a command.
  The induction hypothesis for e_1 and the sub-evaluation $\gamma(e_1) \to^{i'} e_1'$ now gives us that $(n - i', e_1') \in \mu\ Val_{\mu,\rho}$. We also have that $\mathtt{let}(x = e_1')\ \gamma(e_2) = (\mathtt{let}(x = \cdot)\ \gamma(e_2))\langle e_1' \rangle$. Furthermore, if $i'' \leq n - i'$, $(i'', v \in Val_{\mu,\rho})$ then $\mathtt{let}(x = v)\ \gamma(e_2) \to \gamma'(e_2)$ with $\gamma' = \gamma[x \mapsto v]$. It follows from uniformity that $(i'' - 1, \gamma') \in [\![\Gamma, x]\!]_{\mu,\rho}$ and by the induction hypothesis, we have that $(i'' - 1, \gamma'(e_2)) \in \mathcal{E}[\mu\ Val_{\mu,\rho}]$. By Lemma 25, this implies that $(i'', \mathtt{let}(x = v)\ \gamma(e_2)) \in \mathcal{E}[\mu\ Val_{\mu,\rho}]$. We can now apply Axiom A-BIND to obtain that $(n - i', \mathtt{let}(x = e_1')\ \gamma(e_2)) \in \mathcal{E}[\mu\ Val_{\mu,\rho}]$.
  Now take the further evaluation (if any) $\mathtt{let}(x = e_1')\ \gamma(e_2) \to^{i-i'} cmd$. It follows directly that $cmd \in \mu\ Val_{\mu,\rho}$ as required.
- $\Gamma \vdash \mathtt{if}(e_1)\{e_2\}\ \mathtt{else}\ \{e_3\}$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$, $\Gamma \vdash e_3$. Similar to case 3.
- $\Gamma \vdash e_1; e_2$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. Similar to case 3.

- $\Gamma \vdash \texttt{while}(e_1)\{e_2\}$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. We have that $\gamma(\texttt{while}(e_1)\{e_2\}) = \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}$. We will show that this expression is n-acceptable in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $i \le n$ and $\texttt{while}(\gamma(e_1))\{\gamma(e_2)\} \to^i cmd$. This evaluation must factor as follows:

$$\texttt{while}\,(\gamma(e_1))\{\gamma(e_2)\} \to$$
$$\texttt{if}\,(e_1)\{e_2;\texttt{while}(e_1)\{e_2\}\}\ \texttt{else}\,\{\texttt{undef}\}$$
$$\to^{i-1} cmd.$$

Now take the largest prefix of this evaluation such that it is of the form

$$\texttt{if}\,(e_1)\{e_2;\texttt{while}(e_1)\{e_2\}\}\ \texttt{else}\,\{\texttt{undef}\} \to^{i'}$$
$$\texttt{if}\,(e_1)\{e_2;\texttt{while}(e_1)\{e_2\}\}\ \texttt{else}\,\{\texttt{undef}\}.$$

The rest of this case proceeds like case 3.
- $\Gamma \vdash e_1.\texttt{print}\,(e_2)$ with $\Gamma \vdash e_1$, $\Gamma \vdash e_2$. We have that $\gamma(e_1.\texttt{print}\,(e_2)) = \gamma(e_1).\texttt{print}\,(\gamma(e_2))$. We will show that this expression is n-acceptable in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $i \le n$ and $\gamma(e_1).\,\texttt{print}\,(\gamma(e_2)) \to^i cmd$. Take the longest prefix such that $\gamma(e_1).\,\texttt{print}\,(\gamma(e_2)) \to^{i'} e_1'.\,\texttt{print}\,(\gamma(e_2))$. Clearly, $i' \le i$. If the evaluation stops there, then e_1' has to be a command because of Lemma 10. If not, e_1' must be a value by Lemma 13, so also a command. The sub-evaluation $\gamma(e_1) \to^{i'} e_1'$ together with the induction hypothesis then give us that $(n - i', e_1') \in \mu\ \ Val_{\mu,\rho}$. By Lemma 25 and Axiom A-BIND, it now suffices to prove that for $i'' \le n - i'$, $(i'', v) \in Val_{\mu,\rho}$, $(i'', v.\,\texttt{print}\,(\gamma(e_2))) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
So, assume that $j \le i''$, $v.\texttt{print}\,(\gamma(e_2)) \to^j cmd$. Take the longest prefix such that $v.\texttt{print}\,(\gamma(e_2)) \to^{j'} v.\texttt{print}\,(e_2')$. Clearly, $j' \le j$. If the evaluation stops there, then e_2' has to be a command because of Lemma 10. If not, e_2' must be a value by Lemma 13, so also a command. The sub-evaluation $\gamma(e_2) \to^{j'} e_2'$ together with the induction hypothesis and uniformity then give us that $(i'' - j', e_2') \in \mu\ \ Val_{\mu,\rho}$. By Axiom A-BIND and the remaining execution, it now suffices to prove that for $j'' \le i'' - j'$, $(j'', v') \in Val_{\mu,\rho}$, $(j'', v.\,\texttt{print}\,(v')) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
By Lemma 24, it suffices to prove that $v.\,\texttt{print}\,(v') \in \mu\ \ Val_{\mu,\rho}$. Axiom A-PRINT tells us immediately that this is true.
- $\Gamma \vdash \lambda x.e$ with $\Gamma, x \vdash e$. We have that $\gamma(\lambda x.e) = \lambda x.\gamma(e)$. We will show that this expression is n-acceptable in $Val_{\mu,\rho}$. More specifically, we will show that it is in $(Val_{\mu,\rho} \to \mu\ \ Val_{\mu,\rho}) \subseteq Val_{\mu,\rho}$. To do that, we assume $i < n$ and $(i, v) \in Val_{\mu,\rho}$ and we prove that $(i, \gamma(e)[x/v]) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$. We now have that $\gamma(e)[x/v] = \gamma'(e)$ with $\gamma' = \gamma[x/v]$. Our remaining proof obligation then follows directly from the induction hypothesis, since from uniformity, we directly get that $(i, \gamma) \in [\![\Gamma, x]\!]_{\mu,\rho}$.
- $\Gamma \vdash e_1\ e_2$ with $\Gamma \vdash e_1$ and $\Gamma \vdash e_2$. We have that $\gamma(e_1\ e_2) = \gamma(e_1)\ \gamma(e_2)$. We will show that this expression is n-acceptable in $\mathcal{E}[\mu\ \ Val_{\mu,\rho}]$, so assume that $i \le n$ and $\gamma(e_1)\ \gamma(e_2) \to^i cmd$. Take the longest prefix such that $\gamma(e_1)\ \gamma(e_2) \to^{i'} e_1'\ \gamma(e_2)$. Clearly, $i' \le i$. If the evaluation stops there, then e_1' has to be a command because of Lemma 10. If not, e_1' must be a value by Lemma 13, so also a command. The sub-evaluation $\gamma(e_1) \to^{i'} e_1'$ together with the induction hypothesis then give us that $(n - i', e_1') \in \mu\ \ Val_{\mu,\rho}$. By Lemma 25 and Axiom A-BIND, it now suffices to prove that for $i'' \le n - i'$, $(i'', v) \in Val_{\mu,\rho}$, $(i'', v\ \gamma(e_2)) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
So, assume that $j \le i''$, $v\ \gamma(e_2) \to^j cmd$. Take the longest prefix such that $v\ \gamma(e_2) \to^{j'} v\ e_2'$. Clearly, $j' \le j$. If the evaluation stops there, then e_2' has to be a command because of Lemma 10. If not, e_2' must be a value by Lemma 13, so also a command. The sub-evaluation $\gamma(e_2) \to^{j'} e_2'$ together with the induction hypothesis and uniformity then give us that $(i''-j', e_2') \in \mu\ \ Val_{\mu,\rho}$. By Axiom A-BIND and the remaining execution, it now suffices to prove that for $j'' \le i''-j'$, $(j'', v') \in Val_{\mu,\rho}$, $(j'', v\ v') \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$.
So, assume that $k \le j''$ and $v\ v' \to^k cmd$. Since $v\ v'$ is not a command, k must be non-zero and the evaluation must start with an application of evaluation rule E-APP, which implies that $v = \lambda x.e_3$: $(\lambda x.e_3)\ v' \to e_3[x/v'] \to^{k-1} cmd$. The fact that $(i'', v) \in Val_{\mu,\rho}$ together with uniformity, the definition of $Val_{\mu,\rho}$ and Axiom A-RHO-OUT now implies that $(j'', \lambda x.e_3) \in Val_{\mu,\rho} \to \mu\ \ Val_{\mu,\rho}$. Applying the definition of $Val_{\mu,\rho} \to \mu\ \ Val_{\mu,\rho}$ with the fact that $(j'', v') \in Val_{\mu,\rho}$ and uniformity gives us that $(j'' - 1, e_3[x/v']) \in \mathcal{E}[\mu\ \ Val_{\mu,\rho}]$. The remaining execution $e_3[x/v'] \to^{k-1} cmd$ then gives us that $(j'' - 1, cmd) \in \mu\ \ Val_{\mu,\rho}$ as required.

□

Lemma 26. *If* $(n, e) \in \mathcal{E}[IO_{upp}\ \ P]$, $i \le n$ *and* $e \to^i_o v$, *then* o *is in upper case and* $(n - i, v) \in P$.

Proof. Take the longest prefix of the evaluation consisting of *pure* steps: $e \to^{i'} e' \to^{i''}_o v$ with $i' + i'' = i$. Then either
- $i'' = 0$, $i' = i$, $e' = v$ and $o = []$. Then v is also a command, so we get that $(n - i, v) \in IO_{upp}\ \ P$ and by definition of IO_{upp} (with $i = 0$), we get that $(n - i, v) \in P$.
- $i'' > 0$, in which case e' must be a command. Therefore, $(n - i', e') \in IO_{upp}\ \ P$ and the remaining impure execution gives us that o is in upper case and $(n - i' - i'', v) = (n - i, v) \in P$.

□

The following Lemma was used in the proof of Example 2.

Lemma 27. *(IO_{upp}, Ref_{upp}) defined in Example 2 is a valid effect interpretation.*

Proof. For reference, we repeat the definition of IO_{upp} and Ref_{upp}:

$$Ref_{upp} \overset{\mathsf{def}}{=} \emptyset$$
$$IO_{upp}\ P \overset{\mathsf{def}}{=} \left\{ (n, e) \middle| \begin{array}{l} \text{For } i \leq n, \text{ if } e \to^i_o v \text{ then } o \text{ is} \\ \quad \text{in upper case and } (n-i, v) \in P \end{array} \right\}$$

Ref_{upp} is clearly uniform. If P is uniform, then so is $IO_{upp}\ P$: Take $(n, e) \in IO_{upp}\ P$ and assume that $k \leq n$. We show that $(k, e) \in IO_{upp}\ P$. Take $i \leq k$ and assume that $e \to^i_o v$. Clearly, $i \leq k \leq n$, so that we get that o is in upper case and $(n-i, v) \in P$. But $n - i \leq k - i$, so uniformity of P gives us that $(k-i, v) \in P$.

IO_{upp} is also non-expansive: Take $P, Q \in UPred(Val)$ and we prove that $d(IO_{upp}\ P, IO_{upp}\ Q) \leq d(P, Q)$. Because $UPred(A)$ is bisected, it suffices to show that if $d(P, Q) \leq 2^{-n}$ for some n, then also $d(IO_{upp}\ P, IO_{upp}\ Q) \leq 2^{-n}$. By Lemma 39, it suffices to show $(IO_{upp}\ P)_{[n]} = (IO_{upp}\ Q)_{[n]}$ or by symmetry $(IO_{upp}\ P)_{[n]} \subseteq (IO_{upp}\ Q)_{[n]}$. So, take $(k, e) \in (IO_{upp}\ P)_{[n]}$. Then clearly $k < n$. We prove that $(k, e) \in IO_{upp}\ Q$: Assume $i \leq k$ and $e \to^i_o v$. Then we know that o is in uppercase and $(k-i, v) \in P$. But Lemma 39 implies for $d(P, Q) \leq 2^{-n}$ that $P_{[n]} = Q_{[n]}$ and $k - i \leq k < n$, so $(k-i, v)$ is also in Q.

Now it only remains to prove the axioms:

- A-RHO-OUT: $Ref_{upp} \subseteq (\mathbb{N} \times \{\mathtt{out}\})$. Direct.
- A-PURE: For $(n, v) \in P$, (n, v) must be in $IO_{upp}\ P$. Take $i \leq n$ and $v \to^i_o v'$. By Lemma 8, $i = 0$, $v' = v$ and $o = []$. Then clearly, o is in upper case and $(n-i, v') = (n, v) \in P$.
- A-BIND: If $(n, e) \in IO_{upp}\ P$ and $(i, E\langle v\rangle) \in \mathcal{E}[IO_{upp}\ P']$ for all $i \leq n$ and values $(i, v) \in P$, then $(n, E\langle e\rangle) \in \mathcal{E}[IO_{upp}\ P']$. Assume $i \leq n$ and $E\langle e\rangle \to^i cmd$. Then we need to show that $(n-i, cmd) \in IO_{upp}\ P'$. We know that e is a command, and Lemmas 13 and 11 then imply that either e is a value or $i = 0$ and $cmd = E\langle e\rangle$.
  If e is a value, then the definition of IO_{upp} implies (with $i = 0$ that $(n, e) \in P$. The assumption about E for $i = n$ and $v = e$ then gives us that $(n, E\langle e\rangle) \in \mathcal{E}[IO_{upp}\ P']$.
  Otherwise, $i = 0$ and $E\langle e\rangle$ is a command. We then prove that $(n, E\langle e\rangle) \in IO_{upp}\ P'$. So, take $i' \leq n$ and $E\langle e\rangle \to^{i'}_o v$. Take the longest prefix of this evaluation of the form $E\langle e\rangle \to^{i''}_{o'} E\langle e'\rangle \to^{i'''}_{o''} v$. If the evaluation ends there ($i''' = 0$), then $E\langle e'\rangle$ can only be a value if $E = \cdot$ and e' is a value. If not, then by Lemma 14, e' must be a value. In both cases, the sub-evaluation $e \to^{i'}_{o'} e'$ and the assumption about e tells us that o' is in upper case and $(n-i', e') \in P$.
  The assumption about E then tells us that $(n-i', E\langle e'\rangle) \in \mathcal{E}[IO_{upp}\ P']$. The remaining execution $E\langle e'\rangle \to^{i'''}_{o''} v$ with $i'' + i''' = i'$ and $o' +\!\!+ o'' = o$ then tells us by Lemma 26 that o'' is also in upper case and $(n - i'' - i''', v) = (n-i, v) \in P'$.
- A-PRINT: If $(n, v) \in Val_{IO_{upp}, Ref_{upp}}$, then $(n, v.\,\mathtt{print}\,(v')) \in IO_{upp}\ Val_{IO_{upp}, Ref_{upp}}$. Take $i \leq n$ and $v.\,\mathtt{print}\,(v') \to^i_o v''$. Then clearly, $i = 1$, $v = \mathtt{out}$, $v' = str$, $v'' = \mathtt{undef}$, and $o = [str]$. However, $(n, \mathtt{out})$ cannot possibly be in $Val_{IO_{upp}, Ref_{upp}}$ because

$$Val_{IO_{upp}, Ref_{upp}} = (\mathbb{N} \times PureVal) \cup Ref_{upp} \cup$$
$$(Val_{IO_{upp}, Ref_{upp}} \to IO_{upp}\ \ Val_{IO_{upp}, Ref_{upp}})$$

  and $Ref_{upp} = \emptyset$ and $\mathtt{out}$ is clearly not a lambda or element of $PureVal$.

□

Note: the simple lemma mentioned in the proof of Example 2 is Lemma 26.

Appendix C. Well-scopedness judgement definition

The λ_{JS} well-scopedness judgement referred to in Section 3.2, is defined in Figure 9.

Appendix D. Lemmas about operational semantics of λ_{JS}

This section contains some lemmas about the operational semantics of λ_{JS}, that we will need further on.

Lemma 28. *If $v \in Val$ then $v \not\to$ and $(\sigma, v) \not\to$ for all σ.*

Proof. Case analysis. □

Lemma 29. *If $E\langle e\rangle \in Val$, then $e \in Val$.*

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2 \quad \Gamma;\Sigma\vdash e_3}{\Gamma;\Sigma\vdash e_1[e_2] = e_3}\ \text{(WF-UPDATEFIELD)} \qquad \frac{\Gamma, x_1\cdots x_n;\Sigma\vdash e}{\Gamma;\Sigma\vdash \texttt{func}(x_1\cdots x_n)\{\texttt{return}\ e\}}\ \text{(WF-FUNC)}$$

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash \texttt{delete}\ e_1[e_2]}\ \text{(WF-DELETEFIELD)} \qquad \frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash e_1[e_2]}\ \text{(WF-GETFIELD)} \qquad \Gamma;\Sigma\vdash c\ \text{(WF-CONSTANT)}$$

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash e_1 = e_2}\ \text{(WF-ASSIGN)} \qquad \frac{\Gamma;\Sigma\vdash e \quad \overline{\Gamma;\Sigma\vdash e_{arg}}}{\Gamma;\Sigma\vdash e\,(\overline{e_{arg}})}\ \text{(WF-APP)} \qquad \frac{x\in\Gamma}{\Gamma;\Sigma\vdash x}\ \text{(WF-VAR)}$$

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2 \quad \Gamma;\Sigma\vdash e_3}{\Gamma;\Sigma\vdash \texttt{if}(e_1)\{e_2\}\ \texttt{else}\ \{e_3\}}\ \text{(WF-IF)} \qquad \frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma, x;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash \texttt{let}(x = e_1)e_2}\ \text{(WF-LET)} \qquad \frac{\overline{\Gamma;\Sigma\vdash e}}{\Gamma;\Sigma\vdash \{\overline{str : e}\}}\ \text{(WF-REC)}$$

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash \texttt{while}(e_1)\{e_2\}}\ \text{(WF-WHILE)} \qquad \frac{\Gamma;\Sigma\vdash e_1 \quad \Gamma;\Sigma\vdash e_2}{\Gamma;\Sigma\vdash e_1; e_2}\ \text{(WF-SEQ)} \qquad \frac{\Gamma;\Sigma\vdash e}{\Gamma;\Sigma\vdash \texttt{ref}\ e}\ \text{(WF-REF)}$$

$$\frac{\Gamma;\Sigma\vdash e_1 \quad \cdots \quad \Gamma;\Sigma\vdash e_n}{\Gamma;\Sigma\vdash op_n(e_1\cdots e_n)}\ \text{(WF-OP)} \qquad \frac{\Gamma;\Sigma\vdash e}{\Gamma;\Sigma\vdash \texttt{deref}\ e}\ \text{(WF-DEREF)} \qquad \frac{l\in\Sigma}{\Gamma;\Sigma\vdash l}\ \text{(WF-LOC)}$$

Figure 9. Well-formedness judgement $\Gamma;\Sigma\vdash e$ for λ_{JS}.

Proof. Induction on E: the only possible case for E is that $E = \cdot$, $E = \{str : v, \cdots, str : \cdot\}$, or a composition of such E's. In all cases, e must be in Val. ☐

Lemma 30. *If $E\langle e\rangle = E'\langle e'\rangle$, then one of the following must hold:*

- $e \in Val$.
- $e' \in Val$.
- $E = E'\langle E''\rangle$ *with* $E'' \neq \cdot$.
- $E' = E\langle E''\rangle$.

Proof. Induction on E. ☐

Lemma 31. *If $E\langle e\rangle \in Cmd$, then $e \in cmd$.*

Proof. By definition, we have that either

- $E\langle e\rangle$ is a value, but then Lemma 29 says that $e \in Val \subseteq Cmd$.
- $E\langle e\rangle = E'\langle cmd_0\rangle$. By Lemma 30, we then have one of the following cases:
  - $e \in Val \subseteq Cmd$.
  - $cmd_0 \in Val$, but this is not possible.
  - $E = E'\langle E''\rangle$ with $E'' \neq \cdot$. Then $E\langle e\rangle = E'\langle E''\langle e\rangle\rangle = E'\langle cmd_0\rangle$, so that $E''\langle e\rangle = cmd_0$, but by simple case analysis on cmd_0, it's clear that $e \in Val$.
  - $E' = E\langle E''\rangle$. Then $E\langle e\rangle = E'\langle cmd_0\rangle = E\langle E''\langle cmd_0\rangle\rangle$, so that $e = E''\langle cmd_0\rangle$, so that e is directly a command.

☐

Lemma 32. *If $E\langle e_1\rangle \to e'$, then either $e_1 \in Val$ or $e' = E\langle e_2\rangle$ and $e_1 \to e_2$.*

Proof. Consider the judgement $E\langle e_1\rangle \to e'$. By definition, there must be E', e'_1 and e'_2 such that $E\langle e_1\rangle = E'\langle e'_1\rangle$, $e' = E'\langle e'_2\rangle$, $e'_1 \hookrightarrow e'_2$. We know that $e'_1 \notin Val$ because $e'_1 \hookrightarrow e'_2$. By lemma 30, $E\langle e_1\rangle = E'\langle e'_1\rangle$ implies that one of the following cases must hold:

- $e_1 \in Val$.
- $e'_1 \in Val$. Not possible because $e'_1 \hookrightarrow e'_2$ (simple case analysis).
- $E = E'\langle E''\rangle$ with $E'' \neq \cdot$: We then have that $E'\langle E''\langle e_1\rangle\rangle = E\langle e_1\rangle = E'\langle e'_1\rangle$, which implies that $e'_1 = E''\langle e_1\rangle$. By case analysis on the judgement $e'_1 \hookrightarrow e'_2$, we can then show that $e_1 \in Val$.
- $E' = E\langle E''\rangle$: In this case, we have that $E\langle e_1\rangle = E'\langle e'_1\rangle = E\langle E''\langle e'_1\rangle\rangle$, which implies that $e_1 = E''\langle e'_1\rangle$. We can take $e_2 = E''\langle e'_2\rangle$ and we have that $e_1 = E''\langle e'_1\rangle \to E''\langle e'_2\rangle = e_2$ and $e' = E'\langle e'_2\rangle = E\langle E''\langle e'_2\rangle\rangle = E\langle e_2\rangle$.

☐

Lemma 33. *If $(\sigma, E\langle e_1\rangle) \to (\sigma', e')$, then either $e_1 \in Val$ or $e' = E\langle e_2\rangle$ and $(\sigma, e_1) \to (\sigma', e_2)$.*

Proof. From the shape of the impure evaluation rules, we know that there is a E' such that $E\langle e_1\rangle = E'\langle e'_1\rangle$, $e' = E'\langle e'_2\rangle$ and either

- $e'_1 \hookrightarrow e'_2$, $\sigma' = \sigma$
- $e'_1 = \texttt{ref}\ v$, $\sigma' = \sigma[l \mapsto v]$, $l \notin \mathrm{dom}(\sigma)$, $e'_2 = l$
- $e'_1 = \texttt{deref}\ l$, $\sigma' = \sigma$, $e'_2 = h(l)$
- $e'_1 = (l = v)$, $\sigma' = \sigma[l \mapsto v]$, $e'_2 = v$.

From Lemma 30, we then know from $E\langle e_1\rangle = E'\langle e'_1\rangle$ that one of the following cases must hold:

- $e_1 \in \mathit{Val}$.
- $e'_1 \in \mathit{Val}$, but this is not possible.
- $E = E'\langle E''\rangle$ with $E'' \neq \cdot$: We then have that $E'\langle E''\langle e_1\rangle\rangle = E\langle e_1\rangle = E'\langle e'_1\rangle$, which implies that $e'_1 = E''\langle e_1\rangle$. By case analysis on the impure and pure (in the case of E-PURE) evaluation rules, we then get that $e_1 \in \mathit{Val}$.
- $E' = E\langle E''\rangle$: In this case, we have that $E\langle e_1\rangle = E'\langle e'_1\rangle = E\langle E''\langle e'_1\rangle\rangle$, which implies that $e_1 = E''\langle e'_1\rangle$. We can take $e_2 = E''\langle e'_2\rangle$ and we have that $(\sigma, e_1) = (\sigma, E''\langle e'_1\rangle) \to (\sigma', E''\langle e'_2\rangle) = (\sigma', e_2)$ and $e' = E'\langle e'_2\rangle = E\langle E''\langle e'_2\rangle\rangle = E\langle e_2\rangle$.

□

Lemma 34. *If $e \in \mathit{Cmd}$ then $e \not\to$.*

Proof. Case analysis. □

Lemma 35. *If $(\sigma, e) \to (\sigma', e')$ then either*

- $e \in (\mathit{Cmd} \setminus \mathit{Val})$.
- $e \in (\mathit{Expr} \setminus \mathit{Cmd})$ *and* $\sigma = \sigma'$ *and* $e \to e'$.

Proof. Suppose that $e \in (\mathit{Expr} \setminus \mathit{Cmd})$. We need to prove that $h = h'$ and $e \to e'$. We do a case analysis on the judgement $(\sigma, e) \to (\sigma', e')$. Case E-PURE is okay: $h = h'$ and $e \to e'$. In the three other cases, $e = E\langle \mathit{cmd}_0\rangle$ with $\mathit{cmd}_0 = \texttt{deref}\ l$, $\mathit{cmd}_0 = (l = v)$ or $\mathit{cmd}_0 = \texttt{ref}\ v$, so $e \in \mathit{Cmd}$, contradicting the assumption. If $e \in \mathit{Cmd}$, then we know by Lemma 28 that $e \notin \mathit{Val}$. □

Lemma 36. *If $(\sigma, e) \to^i (\sigma', e')$. Then we can factor the evaluation judgement into either*

- $e \to^i e'$ *and* $\sigma = \sigma'$.
- $e \to^{i'} e''$ *and* $(\sigma, e'') \to^{i''} (\sigma', e')$ *with* $e'' \in \mathit{Cmd}$, $i' + i'' = i$ *and* $i'' > 0$.

Proof. By induction on i. If $e \in \mathit{Cmd}$, then either

- $i > 0$ and we're in the second case with $i' = 0$, $i'' = i$ and $e'' = e$.
- $i = 0$ and we're in the first case.

So suppose that $e \notin \mathit{Cmd}$. If $i = 0$, then we're easily in the first case. If $i > 0$ then the evaluation factors as $(\sigma, e) \to (\sigma'', e'') \to^{i-1} (\sigma', e')$. Lemma 35 tells us that $\sigma'' = \sigma$ and $e \to e''$. We can apply induction to the second part of the judgement to get that either

- $e'' \to^{i-1} e'$ and $\sigma'' = \sigma'$, in which case we conclude that $e \to^i e'$ and $\sigma = \sigma'$.
- $e'' \to^{i'} e''$ and $(\sigma, e'') \to^{i''} (\sigma', e')$ with $e'' \in \mathit{Cmd}$, $i' + i'' = i - 1$ and $i'' > 0$. Therefore, we can conclude this case with $e \to^{i'+1} e''$ and $(\sigma, e'') \to^{i''} (\sigma', e')$, $e'' \in \mathit{Cmd}$ and $(i' + 1) + i'' = i$.

□

Appendix E. Proofs and Details for Section 3.2

We will need some theory about metric spaces. We refer to Birkedal et al. for a short review of metric space theory [16, §2.1].

Lemma 37. *$\mathit{UPred}(A)$ is a complete bounded ultrametric space when the distance function is defined as follows. For $P \in \mathit{UPred}(A)$, define $P_{[k]} \stackrel{\mathsf{def}}{=} \{(m, \mathit{cmd}) \in P \mid m < k\}$. Then, for $P, Q \in \mathit{UPred}(A)$, define the following metric (i.e. distance function between two predicates):*

$$d(P, Q) \stackrel{\mathsf{def}}{=} \begin{cases} 2^{-\max\{k \mid P_{[k]} = Q_{[k]}\}} & \textit{if } P \neq Q \\ 0 & \textit{otherwise} \end{cases}$$

Proof. This lemma is already mentioned in Birkedal et al. [16]. It can be proven using standard techniques from metric space theory. □

Note that the lemma above does not depend on any properties of the set A, but works for arbitrary sets.

We will also need the same result for the set of all (not necessarily uniform) step-indexed predicates.

Lemma 38. *$\mathit{Pred}(A)$ is a complete bounded ultrametric space when the distance function is defined in the same way as in Lemma 37 for uniform predicates.*

Proof. This lemma can also be proven using standard techniques from metric space theory. □

Lemma 39. *For $P, Q \in \mathit{UPred}(X)$, we have that $d(P, Q) \leq 2^{-n}$ iff $P =_n Q$ iff $P_{[n]} = Q_{[n]}$.*

Proof. The equivalence between $d(P,Q) \le 2^{-n}$ and $P =_n Q$ is simply by definition. The last equivalence is only slightly harder.

If $P = Q$, there is nothing to prove. If not, then we call $m = \max\{k \mid P_{[k]} = Q_{[k]}\}$ and observe that $d(P,Q) = 2^{-m}$. We have that $d(P,Q) \le 2^{-n}$ iff $m \ge n$ and $m \ge n$ iff $P_{[n]} = Q_{[n]}$. ☐

Lemma 40. *W, Island and StorePred are well-defined.*

Proof. For reference, we repeat the definitions:

$$\begin{aligned}
\text{IslandName} &\overset{\text{def}}{=} \mathbb{N} \\
W &\overset{\text{def}}{=} \{w \in \text{IslandName} \hookrightarrow \text{Island} \mid \text{dom}(w) \text{ finite}\} \\
\text{Island} &\overset{\text{def}}{=} \{\iota = (s, \phi, \phi^{\text{pub}}, H) \mid s \in \text{State} \wedge \phi \subseteq \text{State}^2 \wedge \\
&\qquad \phi^{\text{pub}} \subseteq \phi \wedge \phi, \phi^{\text{pub}} \text{ reflexive and transitive } \wedge \\
&\qquad H \in \text{State} \to \text{StorePred}\} \\
\text{StorePred} &\overset{\text{def}}{=} \{\psi \in \hat{W} \to_{mon,ne} \mathit{UPred}(\text{Store})\} \\
\mathit{roll} &: \frac{1}{2} \cdot W \cong \hat{W}
\end{aligned}$$

We solve this set of (recursive) equations according to the general recipe by Birkedal et al. [16]. This means that we read the equation $\frac{1}{2} \cdot W \approx \hat{W}$ for $\hat{W}$ as a functor F on the category $\mathbf{CBUlt_{ne}}$ of ultrametric spaces and non-expansive functions, mapping ultrametric spaces $\hat{W}$ to other ultrametric spaces $\frac{1}{2} \cdot W$, where W is defined as above in terms of $\hat{W}$. In this case, because $\hat{W}$ occurs (only) contravariantly in the equations, F will actually be a functor from $\mathbf{CBUlt_{ne}}^{op}$ to $\mathbf{CBUlt_{ne}}$. If we can show that this functor F is *locally contractive*, then the America-Rutten theorem (see Birkedal et al. [16]) tells us that there exists a unique (up to isomorphism) solution for the equation and this solution is the ultrametric space that we will refer to as $\hat{W}$. The witness isomorphism is what we call roll.

Note first that we actually need to define how the above definition of W should be seen as an ultrametric space, i.e. what the metric is. For $w, w' \in W$, we define

$$d(w, w') = \begin{cases} 1 & \text{if } \text{dom}(w) \ne \text{dom}(w') \\ \max_j(d(w(j), w'(j))) & \text{otherwise} \end{cases}$$

For $(s_1, \phi_1, \phi_1^{\text{pub}}, H_1), (s_2, \phi_2, \phi_2^{\text{pub}}, H_2) \in \text{Island}$, we define

$$d((s_1, \phi_1, \phi_1^{\text{pub}}, H_1), (s_2, \phi_2, \phi_2^{\text{pub}}, H_2)) = \begin{cases} 1 & \text{if } s_1 \ne s_2 \text{ or } \phi_1 \ne \phi_2 \text{ or } \phi_1^{\text{pub}} \ne \phi_2^{\text{pub}} \\ d(H_1, H_2) & \text{otherwise} \end{cases}$$

The distance between functions $H_1, H_2 \in \text{State} \to \text{StorePred}$ is defined using the uniform metric as usual.

Our F is a locally contractive functor if, for ultrametric spaces $\hat{W}, \hat{W}'$ and non-expansive functions $f, f' : \hat{W}' \to \hat{W}$, we have that $d(F(f), F(f')) \le \phi \cdot d(f, f')$ for some $\phi < 1$. Note that the contravariance of F means that $F(f), F(f') : F(\hat{W}) \to F(\hat{W}')$. We know that $d(F(f), F(f')) = \frac{1}{2} \cdot \sup_w d(F(f)\ w, F(f')\ w)$. So, take $w \in F(\hat{W})$, then it suffices to prove that $d(F(f)\ w, F(f')\ w) \le d(f, f')$ for $\phi = \frac{1}{2}$. Take w, an arbitrary j, $w(j) = (s, \phi, \phi^{\text{pub}}, H)$, an arbitrary $t \in \text{State}$, then it suffices to prove that $d(H\ t \circ f, H\ t \circ f') \le d(f, f')$. This follows from the non-expansiveness of $H\ t$. ☐

Lemma 41. *If $\sigma :_n w$ then for all $n' \le n$ also $\sigma :_{n'} w$.*

Proof. Follows from the definition and the uniformity of StorePreds. ☐

Lemma 42. *Cnst, $\{P\}$, $P \cup P'$ are monotone and uniform Kripke step-indexed command predicates if P and P' are.*
$[P] \to P'$ is monotone and uniform even if P and P' are not.

Proof. For Cnst, $\{P\}$ and $P \cup P'$: easy by direct application of the definition.

For $\mathcal{E}$, monotonicity follows from monotonicity of P and uniformity follows from the quantification over $i \le n$ and the uniformity of P.

For $[P] \to P'$, monotonicity follows from the quantification over future worlds $w' \sqsupseteq w$ and uniformity follows from the quantification over $i < n$. ☐

Lemma 43. $d(\mathcal{E}[P_1], \mathcal{E}[P_2]) \le d(P_1, P_2)$.

Proof. Follows (informally) from the fact that the definition of $\mathcal{E}[P]$ defines when (n, e) is in $\mathcal{E}[P]$ by only looking at (n', e') in P for $n' \le n$. The proof is an easy exercise using Lemma 39. ☐

Lemma 44. *$\{\_\}$ is contractive: $d(\{P\}, \{Q\}) \le \frac{1}{2} \cdot d(P, Q)$ for all $P, Q \in T$.*

Proof. Follows (informally) from the fact that the definition of $\{P\}$ defines when (n, e) is in $\{P\}$ by only looking at (n', e') in P for $n' < n$. The proof is an easy exercise using Lemma 39. □

Lemma 45. $d(P_1 \cup \cdots \cup P_n, P'_1 \cup \cdots \cup P'_n) \leq \max_i(d(P_i, P'_i))$.

Note: this result may seem weird, as the order of P_i and P'_i can be chosen arbitrarily. Nevertheless, the lemma holds for any possible order and badly chosen orders just make the result uninteresting.

Proof. It suffices to show for an arbitrary w that if $(P_i\ w)_{[k]} = (P'_i\ w)_{[k]}$, then $((P_1 \cup \cdots \cup P_n)\ w)_{[k]} = ((P'_1 \cup \cdots P'_n)\ w)_{[k]}$. By symmetry, it suffices to show that $((P_1 \cup \cdots \cup P_n)\ w)_{[k]} \subseteq ((P'_1 \cup \cdots P'_n)\ w)_{[k]}$. So take $(n, v) \in ((P_1 \cup \cdots \cup P_n)\ w)_{[k]}$, then $n < k$ and $(n, v) \in P_j\ w$ for some j. Because $(P_j\ w)_{[k]} = (P'_j\ w)_{[k]}$, we have that $(n, v) \in P'_j\ w \subseteq ((P'_1 \cup \cdots \cup P'_n)\ w)$. Finally, because $n < k$, we get $(n, v) \in ((P'_1 \cup \cdots \cup P'_n)\ w)_{[k]}$. □

Lemma 46. $[\_] \to \_$ *is contractive in both inputs:*

$$d([P_1] \to P'_1, [P_2] \to P'_2) \leq \frac{1}{2} \max(d(P_1, P_2), d(P'_1, P'_2))$$

for all $P_1, P'_1, P_2, P'_2 \in T$.

Proof. Because all metric spaces involved are bisected and by Lemma 39, it suffices to prove that if $(P_1\ w)_{[k]} = (P_2\ w)_{[k]}$ and $(P'_1\ w)_{[k]} = (P'_2\ w)_{[k]}$, then $(([P_1] \to P'_1)\ w)_{[k+1]} = (([P_2] \to P'_2)\ w)_{[k+1]}$. By symmetry, it suffices to prove $(([P_1] \to P'_1)\ w)_{[k+1]} \subseteq (([P_2] \to P'_2)\ w)_{[k+1]}$. So, take $(n, \texttt{func}(x_1 \cdots x_k)\{e\}) \in (([P_1] \to P'_1)\ w)_{[k+1]}$. Then we know that $n \leq k$ and for all $v'_1 \cdots v'_k \in \mathit{Val}$, $w' \sqsupseteq w$, $i < n$ with $\overline{(i, v'_j) \in P_1\ w'}$ that $(i, e[x_1/v'_1, \cdots, x_n/v'_k]) \in \mathcal{E}[P'_1]\ w'$. Now take $v'_1 \cdots v'_k \in \mathit{Val}$, $w' \sqsupseteq w$, $i < n$ with $\overline{(i, v'_j) \in P_2\ w'}$. We have $i < n \leq k$, so that $i < k$. We need to prove that $(i, e[x_1/v'_1, \cdots, x_n/v'_k]) \in \mathcal{E}[P'_2]\ w'$. Because $i < k$ and $(P_1\ w)_{[k]} = (P_2\ w)_{[k]}$, we have that $\overline{(i, v'_j) \in P_1\ w'}$. Therefore, $(i, e[x_1/v'_1, \cdots, x_n/v'_k]) \in \mathcal{E}[P'_1]\ w'$. By Lemmas 43 and our assumptions, we have that $d(\mathcal{E}[P'_1], \mathcal{E}[P'_2]) \leq d(P'_1, P'_2) \leq 2^{-k}$, so that (using Lemma 39) $(\mathcal{E}[P'_1]\ w')_{[k]} = (\mathcal{E}[P'_2]\ w')_{[k]}$ and thus $(i, e[x_1/v'_1, \cdots, x_n/v'_k]) \in \mathcal{E}[P'_2]\ w'$. So we can conclude that $(n, \texttt{func}(x_1 \cdots x_k)\{e\}) \in (([P_2] \to P'_2)\ w)_{k+1}$. □

Lemma 47. *If $e \to^i e'$ then $(k - i, e') \in \mathcal{E}[P]\ w$ iff $(k, e) \in \mathcal{E}[P]\ w$.*

Proof. Suppose that $e \to^i e'$. We prove the two directions:

- left to right: Suppose that $(k - i, e') \in \mathcal{E}[P]\ w$. $j \leq k$, $e \to^j e''$ and $e'' \in \mathit{Cmd}$. Because the evaluation is deterministic and because of Lemma 34, we have that $j \geq i$ and that the evaluation judgement $e \to^j e''$ is an extension of the judgement $e \to^i e'$. The extension forms a judgement $e' \to^{j-i} e''$. We have that $j \leq k$, so $j - i \leq k - i$. So we can apply the assumption $(k - i, e') \in \mathcal{E}[P]\ w$ to get that $((k - i) - (j - i), e'') \in P\ w$ and $(k - i) - (j - i) = k - j$ so this is what we need.
- right to left: Suppose that $(k, e) \in \mathcal{E}[P]\ w$. Now suppose that $j \leq k - i$ and $e' \to^j e''$ with $e'' \in \mathit{Cmd}$. We can concatenate the evaluation judgements to get $e \to^{i+j} e''$ and $i + j \leq k$, so we get from $(k, e) \in \mathcal{E}[P]\ w$ that $(k - i - j, e'') \in P\ w$, which is what we need.

□

Lemma 48. $P\ w \subseteq \mathcal{E}[P]\ w$.

Proof. $(k, e) \in P\ w$ implies that $e \in \mathit{Cmd}$ and by Lemma 34, it does not (purely) reduce. The result then follows from the definition of $\mathcal{E}[P]\ w$. □

Lemma 49. *If $P\ w \subseteq P'\ w$ for all w, then $\mathcal{E}[P]\ w \subseteq \mathcal{E}[P']\ w$ for all w.*

Proof. Direct from the definition of $\mathcal{E}[P]\ w$. □

Lemma 50. *If A and B are complete bounded ultrametric spaces, $f_1, f_2 : A \to_{ne} B$, $v_1, v_2 : A$ then $d(f_1\ v_1, f_2\ v_2) \leq \max(d(f_1, f_2), d(v_1, v_2))$.*

Proof. For the uniform distance function, it is direct that $d(f_1\ v_1, f_2\ v_1) \leq d(f_1, f_2)$ and the non-expansiveness of f_2 implies that $d(f_2\ v_1, f_2\ v_2) \leq d(v_1, v_2)$. The result then follows from the ultrametric inequality: $d(f_1\ v_1, f_2\ v_2) \leq \max(d(f_1\ v_1, f_2\ v_1), d(f_2\ v_1, f_2\ v_2))$. □

Lemma 51. *For an effect interpretation (μ, ρ), $\texttt{JSVal}_{\mu,\rho}$ is well-defined, uniform and monotone.*

Note: this construction works independent of whether (μ, ρ) is a valid *effect interpretation, i.e. whether it satisfies the axioms from Section 3.2.*

Proof. We prove that $\texttt{JSVal}_{\mu,\rho}$ can be defined as the unique Banach fixpoint of a contractive function in T, with $T = W \to_{mon,ne} \mathit{UPred}(\mathit{Val})$.

We define:

$$jsvalRec_{\mu,\rho} : T \to T$$
$$jsvalRec_{\mu,\rho}\ s \overset{\text{def}}{=} Cnst \cup \rho \cup ([s] \to\ \ \mu\ s) \cup \{s\}$$

- $jsvalRec_{\mu,\rho}\ s$ is in T if s is in T, i.e. uniform + monotone: follows from the properties of $Cnst$, $\{\_\}$, $\cup$ and $[\_] \to \_$ proved in Lemma 42 and the assumptions on ρ and μ.
- $jsvalRec_{\mu,\rho}$ is contractive. We temporarily abbreviate $jsvalRec_{\mu,\rho}$ to f. Suppose that $s, t \in T$. Then because of Lemma 45, we have that

  $$d(f(s), f(t)) \leq \max\left(d([s] \to\ \ \mu\ s, [t] \to\ \ \mu\ t), d(\{s\}, \{t\})\right)$$

  Now, $\{\_\}$ and $[\_] \to\ \_$ are contractive (by Lemma 44 and Lemma 46) and μ is non-expansive. As a result of all this, we get that $d(f(s), f(t)) \leq \frac{1}{2} d(s,t)$.

□

Lemma 52. *If* $(n, e) \in \mathtt{JSVal}_{\mu,\rho}\ w$ *then we have the following facts:*

- *If* $e = \mathtt{func}(\bar{x})\{\mathtt{return}\ e\}$ *then* $(n,e) \in ([\mathtt{JSVal}_{\mu,\rho}] \to\ \ \mu\ \mathtt{JSVal}_{\mu,\rho})\ w$.
- *If* $e = \{\overline{str : v}\}$ *then* $(n, e) \in \{\mathtt{JSVal}_{\mu,\rho}\}\ w$.
- *If* $e \in Loc$, *then* $(n,e) \in \rho\ w$.

Proof. This follows from the definition of $\mathtt{JSVal}_{\mu,\rho}$, the definitions of $[\_] \to\_\ \_$, $\{\_\}$ and $Cnst$. □

Lemma 53. *In a complete bounded ultrametric space* U*, suppose we have contractive* $f_1, f_2 : U \to U$ *and suppose that* $d(s_1, s_2) \leq m$ *implies that* $d(f_1(s_1), f_2(s_2)) \leq m$*. Then for the respective unique fixpoints* fp_1 *and* fp_2 *of* f_1 *and* f_2*, we have that* $d(fp_1, fp_2) \leq m$.

Proof. Take an arbitrary $p \in U$ and define $x_{1,0} = x_{2,0} = p$. Then define for $j > 0$: $x_{i,j} = f_i(x_{i,j-1})$. We now have that $d(fp_1, fp_2) = d(\lim_j x_{1,j}, \lim_j x_{2,j})$. Now take N such that for all $n \geq N$, $d(fp_1, x_{1,n}) \leq m$ and $d(fp_2, x_{2,n}) \leq m$. Now, because U is ultrametric, we have that

$$\begin{aligned} d(fp_1, fp_2) &\leq \max(d(fp_1, x_{1,n}), d(x_{1,n}, x_{2,n}), d(fp_2, x_{2,n})) \\ &= \max(m, d(x_{1,n}, x_{2,n})) \end{aligned}$$

But by induction, we have that $d(x_{1,n}, x_{2,n}) \leq m$: it is clearly true for $x_{1,0} = x_{2,0}$ and if $d(x_{1,j-1}, x_{2,j-1}) \leq m$, then $d(x_{1,j}, x_{2,j}) = d(f_1(x_{1,j}), f_2(x_{2,j})) \leq m$. Therefore, $d(fp_1, fp_2) \leq m$. □

Lemma 54.

$$d(\mathtt{JSVal}_{\mu_1,\rho_1}, \mathtt{JSVal}_{\mu_2,\rho_2}) \leq \max(\frac{1}{2}\ d(\mu_1, \mu_2), d(\rho_1, \rho_2))$$

Proof. Following Lemma 53, we take $s_1, s_2 : T$ and assume that $d(s_1, s_2) \leq \max(\frac{1}{2}\ d(\mu_1, \mu_2), d(\rho_1, \rho_2))$. Then we need to show that

$$d(jsvalRec_{\mu_1,\rho_1}\ s_1, jsvalRec_{\mu_2,\rho_2}\ s_2) \leq \max\left(\frac{1}{2}\ d(\mu_1, \mu_2), d(\rho_1, \rho_2)\right).$$

We know that

$$jsvalRec_{\mu,\rho}\ s = Cnst \cup \rho \cup ([s] \to\ \ \mu\ s) \cup \{s\}$$

By Lemma 45, this implies

$$d(jsvalRec_{\mu_1,\rho_1}\ s_1, jsvalRec_{\mu_2,\rho_2}\ s_2) \leq \max\left(d(\rho_1, \rho_2), d\left(([s_1] \to \mu_1\ s_1), ([s_2] \to \mu_2\ s_2)\right), d(\{s_1\}, \{s_2\})\right)$$

By Lemma 44 $\{\_\}$ is contractive, so that $d(\{s_1\}, \{s_2\}) \leq \frac{1}{2}\ d(s_1, s_2)$. Lemma 46 says that

$$d([s_1] \to \mu_1\ s_1, [s_2] \to \mu_2\ s_2) \leq \frac{1}{2}\ \max(d(s_1, s_2), d(\mu_1, \mu_2))$$

Taken together, the above implies, as required, that

$$d(jsvalRec_{\mu_1,\rho_1}\ s_1, jsvalRec_{\mu_2,\rho_2}\ s_2) \leq \max(\frac{1}{2}\ d(\mu_1, \mu_2), d(\rho_1, \rho_2)).$$

□

Lemma 55. *The Axiom* A-BIND *is implied by a combination of two alternative axioms that can be easier to prove:*

- A-BindPrime: *If* $(k,e) \in \mu\ P\ w$ *and* $e \notin \mathit{Val}$ *and if* $(j, E\langle v\rangle) \in \mathcal{E}[\mu\ P']\ w'$ *for all* $j \leq k$, $w' \sqsupseteq w$, $(j,v) \in P\ w'$ *and* $v \in \mathit{Val}$, *then* $(k, E\langle e\rangle) \in \mu\ P'\ w$.
- A-InvPure: $(k,e) \in \mu\ P\ w$ *and* $e \in \mathit{Val}$ *implies that* $(k,e) \in P\ w$.

Proof. Assume that $(j, E\langle v\rangle) \in \mathcal{E}[\mu\ P']\ w'$ for all $j \leq k$, $w' \sqsupseteq w$, $(j,v) \in P\ w'$, $v \in \mathit{Val}$ and $(k,e) \in \mu\ P\ w$. We need to prove that $(k, E\langle e\rangle) \in \mathcal{E}[\mu\ P']\ w$. We distinguish two cases:

- $e \in \mathit{Val}$: by axiom A-InvPure, we get that $(k,e) \in P\ w$. Then the assumption implies that $(k, E\langle e\rangle) \in \mathcal{E}[\mu\ P']\ w$ as required.
- $e \notin \mathit{Val}$. Then the axiom A-BindPrime implies that $(k, E\langle e\rangle) \in \mu\ P'\ w$. By Lemma 48, this implies that $(k, E\langle e\rangle) \in \mathcal{E}[\mu\ P']\ w$ as required.

□

Lemma 56. *The axiom* A-Bind *implies a somewhat more directly useful bind property (applicable to all expressions rather than just commands):*
$(j, E\langle v\rangle) \in \mathcal{E}[\mu\ P']\ w'$ *for all* $j \leq k$, $w' \sqsupseteq w$, $(j,v) \in P\ w'$ *and* $v \in \mathit{Val}$ *and* $(k,e) \in \mathcal{E}[\mu\ P]\ w$, *implies that* $(k, E\langle e\rangle) \in \mathcal{E}[\mu\ P']\ w$.

Proof. Suppose $i \leq k$ and $E\langle e\rangle \rightarrow^i e'$ with $e' \in \mathit{Cmd}$. We need to prove that $(k, e') \in \mu\ P'\ w$. Take the largest i' such that this pure evaluation starts with $E\langle e\rangle \rightarrow^{i'} E\langle e''\rangle$. Then the sub-evaluation $e \rightarrow^{i'} e''$ and Lemma 47 implies that $(k-i', e'') \in \mathcal{E}[\mu\ P]\ w$. We now have that either

- $i' = i$, $e' = E\langle e''\rangle$. It is then easy to show that $e' \in \mathit{Cmd}$ implies that $e'' \in \mathit{Cmd}$, so that $(k-i', e'') \in \mu\ P\ w$. This implies that $e'' \in \mathit{Cmd}$ and we can apply Axiom A-Bind to obtain that $(k-i', E\langle e''\rangle) \in \mu\ P'\ w$. By Lemma 48 and Lemma 47, this implies that $(k, E\langle e\rangle) \in \mathcal{E}[\mu\ P']\ w$ as required.
- $i' > i$, in which case Lemma 32 together with the first step of the remaining evaluation $E\langle e''\rangle \rightarrow^{i-i'} e'$ and the fact that the next reduct of this evaluation cannot be of the form $E\langle\_\rangle$, implies that $e'' \in \mathit{Val}$. The definition of $\mathcal{E}[\_]$ then implies that $(k-i', e'') \in \mu\ P\ w$. The Axiom A-Bind then implies that $(k-i', E\langle e''\rangle) \in \mathcal{E}[\mu\ P']\ w$. The remaining evaluation $E\langle e''\rangle \rightarrow^{i-i'} e'$ then implies by Lemma 47 that $(k-i, e') \in \mu\ P'\ w$.

□

Definition 2 (Semantic typing judgement)**.** *Assume a valid effect interpretation* (μ,ρ).
Define $[\![\Sigma]\!]_{\mu,\rho}$:

$[\![\Sigma]\!]_{\mu,\rho} : \mathit{UPred}(W)$

$[\![\Sigma]\!]_{\mu,\rho} \stackrel{\mathsf{def}}{=} \{(k,w) \mid \textit{for all } l \in \Sigma.(k,l) \in \rho\ w\}$

Define $[\![\Gamma]\!]_{\mu,\rho}$:

$[\![\Gamma]\!]_{\mu,\rho} : W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Val}^{\Gamma})$

$[\![\Gamma]\!]_{\mu,\rho}\ w \stackrel{\mathsf{def}}{=} \{(k,\gamma) \mid \forall x \in \Gamma.(k,\gamma(x)) \in \mathtt{JSVal}_{\mu,\rho}\ w\}$

Define $\mu,\rho;\Gamma;\Sigma \vDash e$ *iff for all* $k \geq 0$, γ, w, $(k,w) \in [\![\Sigma]\!]_{\mu,\rho}$ *and* $(k,\gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w$ *implies that* $(k,\gamma(e)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$.

Lemma 57. *If* $(k,w) \in [\![\Sigma]\!]_{\mu,\rho}$, $j \leq k$, $w \sqsubseteq w'$ *then* $(j,w') \in [\![\Sigma]\!]_{\mu,\rho}$.

Proof. Suppose $w \sqsubseteq w'$, $j \leq k$ and $(k,w) \in [\![\Sigma]\!]_{\mu,\rho}$. Take $l \in \Sigma$. We need to show that $(j,l) \in \rho\ w'$. This follows from $(k,l) \in \rho\ w$ and the monotonicity and uniformity of ρ. □

Lemma 58.
- *If* $(k,\gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w$, $w \sqsubseteq w'$, $j \leq k$, *then* $(j,\gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w'$.
- *If* $(k,\gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w$, $(k,v) \in \mathtt{JSVal}_{\mu,\rho}\ w$ *and* $\gamma' = \gamma[x \mapsto v]$, *then* $(k,\gamma') \in [\![\Gamma,x]\!]_{\mu,\rho}\ w$.

Proof.
- This follows from the definition of $[\![\Gamma]\!]_{\mu,\rho}$ and the monotonicity and uniformity of $\mathtt{JSVal}_{\mu,\rho}$.
- Direct from the definition.

□

Lemma 59 (Compatibility for function application)**.** *Take* (μ,ρ) *a valid effect interpretation. If* $(k,e) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$ *and* $\overline{(i,e_{arg}) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'}$ *for all* $i \leq k$, $w' \sqsupseteq w$. *Then* $(k, e(\overline{e_{arg}})) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$.

Proof. By Lemma 56, it suffices to prove that $(k,e) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$ (but this is given) and that for all $j_0 \leq k$, $w_0 \sqsupseteq w$ and $(j_0,v) \in \mathtt{JSVal}_{\mu,\rho}\ w_0$, we have that $(j_0, v(\overline{e_{arg}})) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_0$. Note that by monotonicity and uniformity, we still have that $\overline{(j_0,e_{arg}) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_0}$. Again by Lemma 56, it suffices to prove that $(j_0, e_{arg,1}) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_0$ (but we know this already) and that for all $j_1 \leq j_0$, $w_1 \sqsupseteq w_0$ and $(j_1,v_1) \in \mathtt{JSVal}_{\mu,\rho}\ w_1$, we have that $(j_1, v(v_1, e_{arg,2}, \cdots e_{arg,n})) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_1$. Again by monotonicity and uniformity, we retain that $\overline{(j_1,e_{arg}) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_1}$ and $(j_1,v) \in \mathtt{JSVal}_{\mu,\rho}\ w_1$. We can iterate this argument so that finally, it remains to be proven that for $j_n \leq \cdots \leq j_1 \leq j_0 \leq k$, $w_n \sqsupseteq \cdots \sqsupseteq w_1 \sqsupseteq w_0 \sqsupseteq w$, $(j_n,v_i) \in \mathtt{JSVal}_{\mu,\rho}\ w_n$ and $(j_n,v) \in \mathtt{JSVal}_{\mu,\rho}\ w_n$, we have that $(j_n, v(v_1,\cdots,v_n)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_n$.

Now suppose that $i \le j_n$ and $v(v_1, \cdots, v_n) \rightarrow^i e'$ with $e' \in \mathit{Cmd}$. Then the evaluation must start with an application of E-APP, i.e. $v = \mathtt{func}(x_1, \cdots, x_n)\{\mathtt{return}\ e_{body}\}$ and $v(v_1, \cdots, v_n) \rightarrow e_{body}[x_1/v_1, \cdots x_n/v_n] \rightarrow^{i-1} e'$. By Lemma 52 and $(j_n, v) \in \mathtt{JSVal}_{\mu,\rho}\ w_n$, we know that $(j_n, v) \in ([\mathtt{JSVal}_{\mu,\rho}] \rightarrow \mu\ \mathtt{JSVal}_{\mu,\rho})\ w_n$. Uniformity tells us that $(j_n - 1, v_i) \in \mathtt{JSVal}_{\mu,\rho}\ w_n$, so we can apply the definition of $([\_] \rightarrow \_)$ to obtain that $(j_n - 1, e_{body}[x_1/v_1, \cdots x_n/v_n]) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_n$. The remaining evaluation $e_{body}[x_1/v_1, \cdots x_n/v_n] \rightarrow^{i-1} e'$ then allows us to conclude that $(j_n - i, e') \in \mu\ \mathtt{JSVal}_{\mu,\rho}\ w_n$ as required. ☐

Using the semantic typing judgement $\mu, \rho; \Gamma; \Sigma \vDash e$, we can reformulate the Fundamental Theorem as follows:

Theorem 3. *If* $\Gamma, \Sigma \vdash e$ *then* $\mu, \rho; \Gamma; \Sigma \vDash e$ *for any* μ *and* ρ *that satisfy the axioms in Section 3.2.*
We prove this theorem by induction on the $\Gamma; \Sigma \vdash e$ *well-formedness judgement.*

Proof. Take $k \ge 0$, γ, w such that $(k, w) \in [\![\Sigma]\!]_{\mu,\rho}$ and $(k, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w$.

We need to prove that $(k, \gamma(e)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$. It is also sufficient to prove that $(k, \gamma(e)) \in \mu\ \mathtt{JSVal}_{\mu,\rho}\ w$ (by Lemma 48) or even $(k, \gamma(e)) \in \mathtt{JSVal}_{\mu,\rho}\ w$ if $\gamma(e) \in \mathit{Val}$ (by Axiom A-PURE).

- Case WF-CONSTANT: $\gamma(c) = c \in \mathit{Val}$ and $(k, c) \in \mathit{Cnst}\ w \subseteq \mathtt{JSVal}_{\mu,\rho}\ w$.
- Case WF-FUNC: $\gamma(\mathtt{func}(\bar{x})\{\mathtt{return}\ e\})$ is $\mathtt{func}(\bar{x})\{\mathtt{return}\ \gamma(e)\} \in \mathit{Val}$. It suffices to show that

  $$(k, \mathtt{func}(\bar{x})\{\mathtt{return}\ \gamma(e)\}) \in ([\mathtt{JSVal}_{\mu,\rho}] \rightarrow\ \mu\ \mathtt{JSVal}_{\mu,\rho})\ w \subseteq \mathtt{JSVal}_{\mu,\rho}\ w.$$

  To prove this, take arbitrary $w' \sqsupseteq w$, $\overline{v_{arg}}$, $i < k$ such that $\overline{(i, v_{arg}) \in \mathtt{JSVal}_{\mu,\rho}\ w'}$. It suffices to show that $(i, \gamma(e)[\overline{x/v_{arg}}])$ is in $\mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$. This follows from the induction hypothesis because
  - $(i, w') \in [\![\Sigma]\!]_{\mu,\rho}$: by the fact that $(k, w) \in [\![\Sigma]\!]_{\mu,\rho}$ and Lemma 57.
  - $\gamma(e)[\overline{x/v_{arg}}] = \gamma'(e)$ with $\gamma' = \gamma[\overline{x \mapsto v_{arg}}]$
  - $(i, \gamma') \in [\![\Gamma, \bar{x}]\!]_{\mu,\rho}\ w'$: we know by Lemma 58 that $(i, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w'$. The fact that $\overline{(i, v_{arg}) \in \mathtt{JSVal}_{\mu,\rho}\ w'}$ gives us the rest.
- Case WF-LOC: We know that $l \in \Sigma$. $\gamma(l) = l \in \mathit{Val}$. The definition of $[\![\Sigma]\!]_{\mu,\rho}$ gives us that $(k, l) \in \rho\ w$, which is contained in $\mathtt{JSVal}_{\mu,\rho}\ w$.
- Case WF-VAR: we know that $(k, \gamma(x)) \in \mathtt{JSVal}_{\mu,\rho}\ w$ by the definition of $[\![\Gamma]\!]_{\mu,\rho}\ w$ and $\gamma(x)$ is a value.
- Case WF-LET: $\gamma(\mathtt{let}(x = e_1)\ e_2) = \mathtt{let}(x = \gamma(e_1))\ \gamma(e_2)$. By Lemma 56, it suffices to prove that
  - $(k, \gamma(e_1)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$
  - $(j, \mathtt{let}(x = v)\ \gamma(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$ for all $j \le k$, $w' \sqsupseteq w$, $v \in \mathit{Val}$ and $(j, v) \in \mathtt{JSVal}_{\mu,\rho}\ w'$.

  The first follows directly from the induction hypothesis. For the second, we have that $\mathtt{let}(x = v)\ \gamma(e_2) \rightarrow \gamma(e_2)[x/v]$. Observe that $\gamma(e_2)[x/v] = \gamma'(e_2)$ with $\gamma' = \gamma[x \mapsto v]$. We know by Lemma 57 that $(j - 1, \gamma') \in [\![\Gamma, x]\!]_{\mu,\rho}\ w'$ and by Lemma 58 that $(j - 1, w') \in [\![\Sigma]\!]_{\mu,\rho}$. The induction hypothesis then gives us that $(j - 1, \gamma'(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$ and by Lemma 47 also $(j, \mathtt{let}(x = v)\ \gamma(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$ as required.
- Case WF-APP: $\gamma(e(\overline{e_{arg}})) = \gamma(e)(\overline{\gamma(e_{arg})})$. The induction hypotheses together with Lemma 59 and Lemmas 57 and 58 tell us that $(k, \gamma(e)(\overline{\gamma(e_{arg})})) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$, as required.
- Case WF-GETFIELD: $\gamma(e_1[e_2]) = \gamma(e_1)[\gamma(e_2)]$. By Lemma 56, it suffices to prove that $(k, \gamma(e_1)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$ (but this follows from the induction hypothesis for e_1) and that for any $j \le k$, $w' \sqsupseteq w$ and $(j, v) \in \mathtt{JSVal}_{\mu,\rho}\ w'$, we have that $(j, v[\gamma(e_2)]) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$. By Lemma 57, we have that $(j, w') \in [\![\Sigma]\!]_{\mu,\rho}$ and by Lemma 58, we have that $(j, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w'$. Again by Lemma 56, it suffices to prove that $(j, \gamma(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w'$ (but this follows again from the induction hypothesis) and that for all $j' \le j$, $w'' \sqsupseteq w'$ and $(j', v') \in \mathtt{JSVal}_{\mu,\rho}\ w''$, we have that $(j', v[v']) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w''$. Uniformity and monotonicity of $\mathtt{JSVal}_{\mu,\rho}$ gives us that also $(j', v) \in \mathtt{JSVal}_{\mu,\rho}\ w''$. So, assume that $j'' \le j'$ and $v[v'] \rightarrow^{j''} e'$ with $e' \in \mathit{Cmd}$. Then we must have that the first step of the evaluation is an application of either E-GETFIELD or E-GETFIELD-NOTFOUND, which implies that $v = \{\overline{str : v''}\}$, $v[v'] \rightarrow e'' \rightarrow^{j''-1} e'$ and either
  - $v' = str_i$ and $e'' = v''_i$: it follows from Lemma 52 that $(j', v) \in \{\mathtt{JSVal}_{\mu,\rho}\}\ w''$, which implies that $(j' - 1, v''_i) \in \mathtt{JSVal}_{\mu,\rho}\ w''$. By Lemma 34, $j'' - 1 = 0$ and $e' = v''_i$. From Axiom A-PURE, we can now conclude that $(j' - j'', e') = (j' - 1, v''_i) \in \mu\ \mathtt{JSVal}_{\mu,\rho}\ w''$.
  - $v' \notin \{\overline{str}\}$ and $e'' = \mathtt{undef}$: in this case, $j'' - 1 = 0$ and $e' = e'' = \mathtt{undef}$. We directly have that $(j' - j'', e') \in \mathit{Cnst} \subseteq \mathtt{JSVal}_{\mu,\rho}\ w''$ and by Axiom A-PURE, this implies that $(j' - j'', e') \in \mu\ \mathtt{JSVal}_{\mu,\rho}\ w''$.
- Case WF-UPDATEFIELD: $(\gamma(e_1[e_2] = e_3)) = (\gamma(e_1)[\gamma(e_2)] = \gamma(e_3))$. Similar to case WF-GETFIELD.
- Case WF-DELETEFIELD: Similar to case WF-GETFIELD.
- Case WF-REC: $\gamma(\{\overline{str : e}\}) = \{\overline{str : \gamma(e)}\}$. Similar to case WF-GETFIELD.
- Case WF-ASSIGN: $(\gamma(e_1 = e_1)) = (\gamma(e_1) = \gamma(e_2))$. By Lemma 56, it suffices to prove that $(k, \gamma(e_1)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w$ (but this follows from the induction hypothesis for e_1) and that for all $j_1 \le k$, $w_1 \sqsupseteq w$, $(j_1, v_1) \in \mathtt{JSVal}_{\mu,\rho}\ w_1$, we have that $(j_1, v_1 = \gamma(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_1$. By Lemmas 57 and 58, we still have that $(j_1, w_1) \in [\![\Sigma]\!]_{\mu,\rho}$ and $(j_1, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w_1$. Again by Lemma 56, it suffices to prove that $(j_1, \gamma(e_2)) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_1$ (but this follows from the induction hypothesis for e_2) and that for all $j_2 \le j_1$, $w_2 \sqsupseteq w_1$, $(j_2, v_2) \in \mathtt{JSVal}_{\mu,\rho}\ w_2$, we have that $(j_2, v_1 = v_2) \in \mathcal{E}[\mu\ \mathtt{JSVal}_{\mu,\rho}]\ w_2$. Note that by

uniformity and monotonicity, we have that $(j_2, v_1) \in \texttt{JSVal}_{\mu,\rho}\ w_2$. It suffices to show that $(j_2, v_1 = v_2) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w_2$, and this follows directly from Axiom A-ASSIGN.

- Case WF-REF: $\gamma(\texttt{ref}\ e) = \texttt{ref}\ \gamma(e)$. By Lemma 56, it suffices to prove that $(k, \gamma(e)) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$ (but this follows from the induction hypothesis for e) and that for all $j \le k$, $w' \sqsupseteq w$, $(j, v) \in \texttt{JSVal}_{\mu,\rho}\ w'$, we have that $(j, \texttt{ref}\ v) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. This follows from Axiom A-REF which gives us that $(j, \texttt{ref}\ v) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w'$ and Lemma 48.
- Case WF-DEREF: $\gamma(\texttt{deref}\ e) = \texttt{deref}\ \gamma(e)$. By Lemma 56, it suffices to show that $(k, \gamma(e)) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$ (but this follows directly from the induction hypothesis for e) and that for all $j \le k$, $w' \sqsupseteq w$, $(j, v) \in \texttt{JSVal}_{\mu,\rho}\ w'$, we have that $(j, \texttt{deref}\ v) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. By Lemma 48, it suffices to show that $(j, \texttt{deref}\ v) \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w'$, which follows directly from Axiom A-DEREF.
- Case WF-IF: $\gamma(\texttt{if}(e_1)\{e_2\}\ \texttt{else}\ \{e_3\}) = \texttt{if}(\gamma(e_1))\{\gamma(e_2)\}\ \texttt{else}\ \{\gamma(e_3)\}$. By Lemma 56, it suffices to show that $(k, \gamma(e_1)) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$ (but this follows directly from the induction hypothesis for e_1), and that for all $j \le k$, $w' \sqsupseteq w$, $(j, v) \in \texttt{JSVal}_{\mu,\rho}\ w'$, we have that $(j, \texttt{if}(v)\{\gamma(e_2)\}\texttt{else}\{\gamma(e_3)\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. So, assume $i \le j$ and $\texttt{if}(v)\{\gamma(e_2)\}\texttt{else}\{\gamma(e_3)\} \to^i e'$ with $e' \in \mathit{Cmd}$. The first step of the evaluation must be an application of evaluation rule E-IFTRUE or E-IFFALSE, so that $v = \texttt{true}$ or $v = \texttt{false}$ and respectively $\texttt{if}(v)\{\gamma(e_2)\}\ \texttt{else}\ \{\gamma(e_3)\} \to \gamma(e_2) \to^{i-1} e'$ or $\texttt{if}(v)\{\gamma(e_2)\}\ \texttt{else}\ \{\gamma(e_3)\} \to \gamma(e_3) \to^{i-1} e'$. By Lemmas 57 and 58, we have that $(j-1, w') \in [\![\Sigma]\!]_{\mu,\rho}$ and $(j-1, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w'$. By induction, we then have that $(j-1, \gamma(e_2))$ and $(j-1, \gamma(e_3))$ are both in $\mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$ and the remaining evaluations give us that $(j-i, e') \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w'$, as required.
- Case WF-SEQ: $\gamma(e_1; e_2) = \gamma(e_1); \gamma(e_2)$. Similar to case WF-IF.
- Case WF-WHILE: $\gamma(\texttt{while}(e_1)\{e_2\}) = \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}$. We proceed by complete induction on k. Suppose that $i \le k$, $\texttt{while}(\gamma(e_1))\{\gamma(e_2)\} \to^i e'$, $e' \in \mathit{Cmd}$. We need to prove that $(k-i, e') \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w$. The first step in the evaluation must be an application of evaluation rule E-WHILE, i.e.

  $$\texttt{while}(\gamma(e_1))\{\gamma(e_2)\} \to \texttt{if}(\gamma(e_1))\{\gamma(e_2); \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}\}\ \texttt{else}\ \{\texttt{undef}\} \to^{i-1} e'$$

  It is then sufficient to show that

  $$(k-1, \texttt{if}(\gamma(e_1))\{\gamma(e_2); \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}\}\ \texttt{else}\ \{\texttt{undef}\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$$

  By Lemma 56, it is sufficient to show that $(k-1, \gamma(e_1)) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w$ (but this follows from the induction hypothesis for e_1) and that for all $j \le k-1$, $w' \sqsupseteq w$, $(j, v) \in \texttt{JSVal}_{\mu,\rho}\ w'$, we have that $(j, \texttt{if}(v)\{\gamma(e_2); \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}\}\texttt{else}\{\texttt{undef}\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. So suppose that $i' \le j$, $\texttt{if}(v)\{\gamma(e_2); \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}\}\texttt{else}\{\texttt{undef}\} \to^{i'} e''$ with $e'' \in \mathit{Cmd}$. The first step of the evaluation must be an application of evaluation rule E-IFTRUE or E-IFFALSE. In the second case, we have that $i' = 1$, $e'' = \texttt{undef}$ and by the defining equation of $\texttt{JSVal}_{\mu,\rho}$, we have that $(j - i', e'') \in \mathit{Cnst} \subseteq \texttt{JSVal}_{\mu,\rho}\ w'$, as required. In the case that rule E-IFTRUE is used, it suffices to prove that $(j-1, \gamma(e_2); \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. By Lemma 56, it suffices to prove that $(j-1, \gamma(e_2)) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$ (but this follows from the induction hypothesis for e_2) and that for all $j' \le j-1$, $w'' \sqsupseteq w'$, $(j', v') \in \texttt{JSVal}_{\mu,\rho}\ w''$, we have that $(j', v'; \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w''$. So, suppose that $i'' \le j'$ and $v'; \texttt{while}(\gamma(e_1))\{\gamma(e_2)\} \to^{i''} e'''$ with $e''' \in \mathit{Cmd}$. The first step of the evaluation must be an application of evaluation rule E-DISCARD-BEGIN, and it is sufficient to prove that $(j'-1, \texttt{while}(\gamma(e_1))\{\gamma(e_2)\}) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w''$. By Lemmas 57 and 58, we still have that $(j'-1, w'') \in [\![\Sigma]\!]_{\mu,\rho}$ and $(j'-1, \gamma) \in [\![\Gamma]\!]_{\mu,\rho}\ w''$. Finally, this now follows from the induction hypothesis for the k that we started with.
- Case WF-OP: $\gamma(op_n(\overline{e_i})) = op_n\left(\overline{\gamma(e_i)}\right)$. By repeatedly applying Lemma 56 and the induction hypotheses for the $\overline{e_i}$ (as we did above for the case WF-APP) we can show that it is sufficient to prove that for $j \le k$, $w' \sqsupseteq w$, $(j, v_i) \in \texttt{JSVal}_{\mu,\rho}\ w'$, that $(j, op_n(\overline{v_i})) \in \mathcal{E}[\mu\ \texttt{JSVal}_{\mu,\rho}]\ w'$. Now suppose that $j' \le j$ and $op_n(\overline{v_i}) \to^{j'} e'$ with $e' \in \mathit{Cmd}$. The first step in this evaluation must be an application of rule E-PRIM. This implies that $j' = 1$ and $e' = \delta_n(op_n, v_1, \cdots v_n)$ and we easily find that $(j', e') \in \mathit{Cnst}\ w' \subseteq \texttt{JSVal}_{\mu,\rho}\ w'$ and by Axiom A-PURE also $(j', e') \in \mu\ \texttt{JSVal}_{\mu,\rho}\ w'$, as required.

□

Appendix F. Proofs and details for Section 4

Lemma 60. *If $w =_n w'$ then $\sigma :_n w$ iff $\sigma :_n w'$.*

Proof. By symmetry, it suffice to prove the left-to-right implication. We get $\sigma = \uplus_{j \in \mathrm{dom}(w)} \sigma_j$ and for all $j \in \mathrm{dom}(w)$ and $k < n$ $(k, \sigma_j) \in w(j).H(w(j).s)\ (\mathit{roll}\ w)$. By definition of the distance function on worlds, we know that $\mathrm{dom}(w') = \mathrm{dom}(w)$. We choose $\sigma'_j = \sigma_j$ and it suffices to prove that also $(k, \sigma_j) \in w'(j).H(w'(j).s)\ (\mathit{roll}\ w')$. But by definition of the distance function on worlds, $w'(j).s = w(j).s$ and we define for brevity $s = w(j).s$. Furthermore, . We have that

$$d(w'(j).H(s)\ (\mathit{roll}\ w'), w(j).H(s)\ (\mathit{roll}\ w)) \le \begin{array}{r} \max(d(w'(j).H(s)\ (\mathit{roll}\ w'), w'(j).H(s)\ (\mathit{roll}\ w)), \\ d(w'(j).H(s)\ (\mathit{roll}\ w), w(j).H(s)\ (\mathit{roll}\ w))) \end{array}$$

Since $w'(j).H(s)$ is non-expansive and $roll\ w =_{n+1} roll\ w'$, we get that

$d(w'(j).H(s)\ (roll\ w'), w'(j).H(s)\ (roll\ w)) \leq 2^{-n-1}$.

Furthermore, since $w'(j).H =_n w'(j).H$, we also get that

$d(w'(j).H(s)\ (roll\ w), w(j).H(s)\ (roll\ w)) \leq 2^{-n}$

So by the ultrametric inequality,

$d(w'(j).H(s)\ (roll\ w'), w(j).H(s)\ (roll\ w)) \leq 2^{-n}$

This in turn means that $(k, \sigma_j) \in w'(j).H(s)\ (roll\ w')$ iff $(k, \sigma_j) \in w(j).H(s)\ (roll\ w)$ (by Lemma 39) for all j, so we're done. ☐

Lemma 61. *If $w_1 =_n w_2$ and $w_1' \sqsupseteq w_1$ then there exists a w_2' such that $w_2' \sqsupseteq w_2$ and $w_2' =_n w_1'$.*

Proof. Take w_2' such that $\mathrm{dom}(w_2') = \mathrm{dom}(w_1')$ and

$$w_2'(j) \stackrel{\mathsf{def}}{=} \begin{cases} w_2(j) & \text{if } n = 0 \wedge j \in \mathrm{dom}(w_2) \\ (s_1', \phi_1, \phi_1^{\mathrm{pub}'}, H_2) & \text{if } n > 0 \wedge j \in \mathrm{dom}(w_2) \wedge w_1'(j) = (s_1', \phi_1', \phi_1^{\mathrm{pub}'}, H_1') \wedge w_2(j) = (s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2) \\ w_1'(j) & \text{otherwise} \end{cases}$$

If $n = 0$, it is clear that $w_2' \sqsupseteq w_2$ and $w_2' =_n w_1'$, so assume that $n > 0$, so we know that $d(w_1, w_2) \leq 2^{-n} < 1$. We know that $d(w_2', w_1') = \max_j d(w_2'(j), w_1'(j))$. For $j \notin \mathrm{dom}(w_2)$, $d(w_2'(j), w_1'(j))$ is clearly zero. For $j \in \mathrm{dom}(w_2)$, assume that $w_1(j) = (s_1, \phi_1, \phi_1^{\mathrm{pub}}, H_1)$, $w_1'(j) = (s_1', \phi_1', \phi_1^{\mathrm{pub}'}, H_1')$ and $w_2(j) = (s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2)$. We know from $w_1' \sqsupseteq w_1$ that $\phi_1' = \phi_1$, $H_1' = H_1$, $\phi_1' \supset \phi_1^{\mathrm{pub}'} \supseteq \phi_1^{\mathrm{pub}}$ and $(s_1, s_1') \in \phi_1$. We know from $d(w_1, w_2) < 1$ that $s_1 = s_2$, $\phi_1 = \phi_2$, $\phi_1^{\mathrm{pub}} = \phi_2^{\mathrm{pub}}$. Therefore, it is clear that $w_2'(j) \sqsupseteq w_2(j)$, so we can conclude that $w_2' \sqsupseteq w_2$. Furthermore, we have that $d(w_2'(j), w_1'(j)) = d(H_2(j), H_1(j)) \leq 2^{-n}$. ☐

Lemma 62. *If $w_1 =_n w_2$ and $w_1' \sqsupseteq^{\mathrm{pub}} w_1$ then there exists a w_2' such that $w_2' \sqsupseteq^{\mathrm{pub}} w_2$ and $w_2' =_n w_1'$.*

Proof. Take w_2' such that $\mathrm{dom}(w_2') = \mathrm{dom}(w_1')$ and

$$w_2'(j) \stackrel{\mathsf{def}}{=} \begin{cases} w_2(j) & \text{if } n = 0 \wedge j \in \mathrm{dom}(w_2) \\ (s_1', \phi_1, \phi_1^{\mathrm{pub}'}, H_2) & \text{if } n > 0 \wedge j \in \mathrm{dom}(w_2) \wedge w_1'(j) = (s_1', \phi_1', \phi_1^{\mathrm{pub}'}, H_1') \wedge w_2(j) = (s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2) \\ w_1'(j) & \text{otherwise} \end{cases}$$

If $n = 0$, it is clear that $w_2' \sqsupseteq^{\mathrm{pub}} w_2$ and $w_2' =_n w_1'$, so assume that $n > 0$, so we know that $d(w_1, w_2) \leq 2^{-n} < 1$. We know that $d(w_2', w_1') = \max_j d(w_2'(j), w_1'(j))$. For $j \notin \mathrm{dom}(w_2)$, $d(w_2'(j), w_1'(j))$ is clearly zero. For $j \in \mathrm{dom}(w_2)$, assume that $w_1(j) = (s_1, \phi_1, \phi_1^{\mathrm{pub}}, H_1)$, $w_1'(j) = (s_1', \phi_1', \phi_1^{\mathrm{pub}'}, H_1')$ and $w_2(j) = (s_2, \phi_2, \phi_2^{\mathrm{pub}}, H_2)$. We know from $w_1' \sqsupseteq^{\mathrm{pub}} w_1$ that $\phi_1' = \phi_1$, $H_1' = H_1$, $\phi_1^{\mathrm{pub}'} = \phi_1^{\mathrm{pub}}$ and $(s_1, s_1') \in \phi_1^{\mathrm{pub}}$. We know from $d(w_1, w_2) < 1$ that $s_1 = s_2$, $\phi_1 = \phi_2$, $\phi_1^{\mathrm{pub}} = \phi_2^{\mathrm{pub}}$. Therefore, it is clear that $w_2'(j) \sqsupseteq^{\mathrm{pub}} w_2(j)$, so we can conclude that $w_2' \sqsupseteq w_2$. Furthermore, we have that $d(w_2'(j), w_1'(j)) = d(H_2(j), H_1(j)) \leq 2^{-n}$. ☐

Lemma 63. *$IO^{std}\ P\ w$ is non-expansive in P and w.*

Proof. From the definition of IO^{std}, it follows easily that $P =_n P'$ implies $IO^{std}\ P =_n IO^{std}\ P'$. Similarly, if $w =_n w'$ then we can conclude from Lemmas 60, 61 and 62 and the non-expansiveness of P that $IO^{std}\ P\ w =_n IO^{std}\ P\ w'$. ☐

Lemma 64. *Ref^{std} is well-defined, non-expansive and monotone in w.*

Proof. We define Ref^{std} as the fixpoint of a contractive function $Ref^{std,rec}$:

$$\begin{aligned} \iota_l^{std,rec} &: (W \to_{mon,ne} UPred(Loc)) \to \text{Island} \\ \iota_l^{std,rec}\ \rho &\stackrel{\mathsf{def}}{=} (l, =, =, H^{std,rec}\ \rho) \\ H^{std,rec} &: (W \to_{mon,ne} UPred(Loc)) \to Loc \to \text{StorePred} \\ H^{std,rec}\ \rho\ l\ w &\stackrel{\mathsf{def}}{=} \left\{(n, \{l \mapsto v\}) \mid n = 0 \text{ or } (n-1, v) \in \mathtt{JSVal}_{IO^{std},\rho}\ (roll_{\hat{W}}^{-1}\ w)\right\} \\ Ref^{std,rec} &: (W \to_{mon,ne} UPred(Loc)) \to (W \to_{mon,ne} UPred(Loc)) \\ Ref^{std,rec}\ \rho\ w &\stackrel{\mathsf{def}}{=} \left\{(n, l) \mid \exists j.\ w(j) =_{n+1} \iota_l^{std,rec}\ \rho\right\} \end{aligned}$$

We need to prove that

- If $\rho \in (W \to_{mon,ne} UPred(Loc))$, then $H^{std,rec} \rho\ l$ is in $\mathrm{StorePred}$. Take $w_1 =_{n+1} w_2$ in $\hat{W}$. Then $roll_{\hat{W}}^{-1}\ w_1 =_n roll_{\hat{W}}^{-1}\ w_2$ in W. $\mathtt{JSVal}_{IO^{std},\rho}$ is non-expansive, so $\mathtt{JSVal}_{IO^{std},\rho}\ (roll_{\hat{W}}^{-1}\ w_1) =_n \mathtt{JSVal}_{IO^{std},\rho}\ (roll_{\hat{W}}^{-1}\ w_2)$. From this, it follows easily (with Lemma 39) that $H^{std,rec} \rho\ l\ w_1 =_{n+1} H^{std,rec} \rho\ l\ w_2$ as required.

- If $\rho \in (W \rightarrow_{mon,ne} UPred(Loc))$, then so is $Ref^{std,rec}\ \rho$: If $w =_n w'$ then $Ref^{std,rec}\ \rho\ w =_n Ref^{std,rec}\ \rho\ w'$: follows from Lemma 39 and the definitions. If $w' \sqsupseteq w$, then it follows from the definition that $Ref^{std,rec}\ \rho\ w' \supseteq Ref^{std,rec}\ \rho\ w$.
- $Ref^{std,rec}$ is contractive in ρ: take $\rho_1, \rho_2 \in W \rightarrow_{mon,ne} UPred(Loc)$ with $\rho_1 =_n \rho_2$. We know by Lemma 54 that $\texttt{JSVal}_{\mu,\rho}$ is non-expansive in ρ, so $\texttt{JSVal}_{IO^{std},\rho_1} =_n \texttt{JSVal}_{IO^{std},\rho_2}$. From that, it follows easily that $H^{std,rec}\ \rho_1 =_{n+1} H^{std,rec}\ \rho_2$. From that, it follows again easily that $Ref^{std,rec}\ \rho_1 =_{n+1} Ref^{std,rec}\ \rho_2$ as required.

We then get Ref^{std} as the unique Banach fixpoint of $Ref^{std,rec}$. □

Lemma 65. *If $(k,e) \in \mathcal{E}[IO^{std}\ P]\ w$, then for all $i \leq k$, $\sigma_r :_k w$, σ_f with $(\sigma_r \uplus \sigma_f, e) \rightarrow^i (\sigma', e')$ and $e' \in Val$ implies that there exists a σ_r' such that $\sigma' = \sigma_r' \uplus \sigma_f$ and there exists a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $\sigma_r' :_{k-i} w'$ and $(k-i, e') \in P\ w'$.*

Proof. By Lemma 36, the evaluation $(\sigma_r \uplus \sigma_f, e) \rightarrow^i (\sigma', e')$ factors in one of the following ways:

- $e \rightarrow^i e'$, $\sigma_r \uplus \sigma_f = \sigma'$. In this case, we can conclude with $\sigma_r' = \sigma_r$, $w' = w$, since $\sigma_r :_{k-i} w$ because of Lemma 41, $(k-i, e') \in IO^{std}\ P\ w$ by definition of $\mathcal{E}[\_]$ and then $(k-i, e') \in P\ w$ by definition of IO^{std} since e' is a value.
- $e \rightarrow^{i'} e''$ and $(\sigma_r \uplus \sigma_f, e'') \rightarrow^{i''} (\sigma', e')$ with $e'' \in Cmd$, $i' + i'' = i$ and $i'' > 0$. In this case, we get that $(k-i', e'') \in IO^{std}\ P\ w$ by definition of $\mathcal{E}[\_]$. By Lemma 41, we get that $\sigma_r :_{k-i'} w$. Because $i'' > 0$ and $i'' = i - i' \leq k - i'$, the definition of IO^{std} gives us a σ_r' with $\sigma' = \sigma_r' \uplus \sigma_f$ and a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $\sigma_r' :_{k-i} w'$ and $(k-i, e') \in P\ w'$.

□

Lemma 66. *(IO^{std}, Ref^{std}) is a valid effect interpretation, i.e. it satisfies the axioms from Section 3.2.*

Proof. We look at all the axioms in turn. For easy reference, we will always first re-state them with IO^{std} and Ref^{std} filled in for μ and ρ.

We use Lemma 55 and prove alternative axioms A-BINDPRIME and A-INVPURE instead of A-BIND.

- A-BINDPRIME: We need to show that if $(k,e) \in IO^{std}\ P\ w$, $e \notin Val$ and $(i, E\langle v\rangle) \in \mathcal{E}[IO^{std}\ P']\ w'$ for all $i \leq k$, $w' \sqsupseteq w$, $(i,v) \in P\ w'$ then $(k, E\langle e\rangle) \in IO^{std}\ P'\ w$.
  By the definition of IO^{std}, we need to show first that $(k, E\langle e\rangle) \in P'\ w$ if $E\langle e\rangle \in Val$, but this is vacuously true because $E\langle e\rangle \in (Cmd \setminus Val)$ by Lemma 29. Secondly, take $0 < i \leq k$ and assume that $\sigma_r :_k w$, $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow^i (\sigma', e')$ and $e' \in Val$. Then we need to produce a σ_r' with $\sigma' = \sigma_r' \uplus \sigma_f$ and a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $\sigma_r' :_{k-i} w'$ and $(k-i, e') \in P'\ w'$.
  Take i' the largest number such that the evaluation $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow^i (\sigma', e')$ starts with $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow^{i'} (\sigma'', E\langle e''\rangle)$ for some e''. By Lemmas 33 and 29, we know that e'' must be a value. The sub-evaluation $(\sigma_r \uplus \sigma_f, e) \rightarrow^{i'} (\sigma'', e'')$ together with $(k,e) \in IO^{std}\ P\ w$ and the other assumptions above gives us a σ_r'' with $\sigma'' = \sigma_r'' \uplus \sigma_f$ and a $w'' \sqsupseteq^{\mathrm{pub}} w$ such that $\sigma_r'' :_{k-i'} w''$ and $(k-i', e'') \in P\ w''$.
  We can now apply the assumption about E to e'' to conclude that $(k-i', E\langle e''\rangle) \in \mathcal{E}[IO^{std}\ P']\ w''$, since $e'' \in Val$, $w'' \sqsupseteq w$ and $k - i' \leq k$. We can then apply Lemma 65 to the remaining evaluation $(\sigma'', E\langle e''\rangle) \rightarrow^{i-i'} (\sigma', e')$ since $\sigma'' = \sigma_r'' \uplus \sigma_f$, $i - i' \leq k - i'$ and $\sigma_r'' :_{k-i'} w''$ and $e' \in Val$. We obtain a σ_r' such that $\sigma' = \sigma_r' \uplus \sigma_f$ and a $w' \sqsupseteq^{\mathrm{pub}} w''$ such that $\sigma_r' :_{k-i} w'$ and $(k-i, e') \in P\ w'$ as required.
- A-INVPURE: $(k,e) \in IO^{std}\ P\ w$ and $e \in Val$ implies that $(k,e) \in P\ w$. By definition.
- A-PURE: If $v \in Val$, then $(n,v) \in P\ w$ implies $(n,v) \in IO^{std}\ P\ w$. By definition of IO^{std}, we first need to prove that if $v \in Val$, then $(n,v) \in P\ w$, which is obviously fine. Second, assume $0 < i \leq n$, $\sigma_r :_n w$, σ_f, $(\sigma_r \uplus \sigma_f, v) \rightarrow^i (\sigma', e')$ and $e' \in Val$. By Lemma 28, the latter is not possible with $i > 0$, so this requirement is also OK.
- A-ASSIGN: Suppose $(n, e_1) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$, $(n, e_2) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. Then $(n, e_1 = e_2) \in IO^{std}\ \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. First, it's clear that $e_1 = e_2 \notin Val$. Second, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f and $(\sigma_r \uplus \sigma_f, e_1 = e_2) \rightarrow^i (\sigma', e')$ and $e' \in Val$. Because e_1 and e_2 are values, we have that the impure evaluation must start with an application of rule E-SETREF. This further implies that $e_1 = l$, $i = 1$, $e' = e_2$ and $\sigma' = \sigma[l \mapsto e_2]$. By Lemma 52 and monotonicity, we obtain that $(n, e_1) = (n, l) \in Ref^{std}\ w$, which gives us a j such that $w(j) =_{n+1} \iota_l^{std}$. From $\sigma_r :_n w$, we then know that $l \in \mathrm{dom}(\sigma_r)$. We can take $w' = w$, $\sigma_r' = \sigma_r[l \mapsto e_2]$ and it is clear that $w' \sqsupseteq^{\mathrm{pub}} w$, $\sigma' = \sigma_r' \uplus \sigma_f$. From uniformity, we directly have that $(n-i, e_2) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w'$. Finally, since $w(j) =_{n+1} \iota_l^{std}$, $(n-i, e_2) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w'$ by uniformity, and $\sigma_r :_{n-i} w'$ by Lemma 41, we can conclude that $\sigma_r' :_{n-i} w'$.
- A-REF: Suppose $(n, e) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. $(n, \texttt{ref}\ e) \in IO^{std}\ \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. First, it's clear that $\texttt{ref}\ e \notin Val$. Second, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f such that $(\sigma_r \uplus \sigma_f, \texttt{ref}\ e) \rightarrow^i (\sigma', e')$ with $e' \in Val$. Because e is a value, we have that the impure evaluation must start with an application of rule E-REF. This further implies that $e' = l$ for some $l \notin \mathrm{dom}(\sigma)$, $i = 1$, $\sigma' = \sigma[l \mapsto e]$. We have that $l \in \mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)$, so we can take $\sigma_r' = \sigma_r[l \mapsto e]$, and it is clear that $\sigma' = \sigma_r' \uplus \sigma_f$. Furthermore we take $w' = w[j \mapsto \iota_l^{std}]$ for some $j \notin \mathrm{dom}(w)$ and it remains to show that $\sigma_r' :_{n-1} w'$ and $(n-1, l) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w'$. The latter follows because $(n-1, l) \in Ref^{std}\ w'$ by definition and $Ref^{std}\ w' \subseteq \texttt{JSVal}_{IO^{std},Ref^{std}}\ w'$ by the definition of $\texttt{JSVal}_{\mu,\rho}$. The former follows by definition of $\sigma_r' :_{n-1} w'$, by the fact that $\sigma_r' = \sigma_r[l \mapsto e]$, $l \notin \mathrm{dom}(\sigma_r)$, $\sigma_r :_{n-i} w$ (by Lemma 41) and $(n-i, e) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w'$ (by monotonicity and uniformity).
- A-DEREF: Suppose $(n, e) \in \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. Then $(n, \texttt{deref}\ e) \in IO^{std}\ \texttt{JSVal}_{IO^{std},Ref^{std}}\ w$. First, it's clear that $\texttt{deref}\ e \notin Val$. Second, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f and $(\sigma_r \uplus \sigma_f, \texttt{deref}\ e) \rightarrow^i (\sigma', e')$ with $e' \in Val$. Because e is a value,

the impure evaluation must start with an application of rule E-DEREF. This further implies that $e = l$, $i = 1$, $e' = \sigma(l)$ and $\sigma' = \sigma_r \uplus \sigma_f$. By Lemma 52 and monotonicity, we obtain that $(n, l) \in \mathit{Ref}^{std}\ w$, which implies that $w(j) =_{n+1} \iota_l^{std}$ for some j. Together with $\sigma_r :_n w$, this in turn implies that $l \in \mathrm{dom}(\sigma_r)$ and $(n-1, \sigma_r(l)) \in \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w$. We can then take $\sigma_r' = \sigma_r$, $w' = w$ and it is clear that $\sigma' = \sigma_r' \uplus \sigma_f$. It follows from $\sigma :_n w$ and Lemma 41 that $\sigma_r' :_{n-1} w'$ and we already know the fact that $(n-1, \sigma(l)) \in \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'$.

□

Lemma 1. *Take an arbitrary store σ, $\Sigma = \mathrm{dom}(\sigma)$ and a value $\emptyset; \Sigma \vdash \mathit{attacker}$. If $(\sigma, \mathit{ticketDispenserTest}\ \mathit{attacker}) \to (\sigma', v)$, then v is even and ≥ 0.*

Proof. If the evaluation terminates, then for some $l \notin \Sigma$ and for some σ', it must factor as follows (omitting the body of $\mathit{dispTkt}$ for brevity):

$$(\sigma, \mathit{ticketDispenser}\ \ \mathit{attacker}) \to^*$$
$$(\sigma[l \mapsto 0], \mathit{attacker}\ \ (\mathtt{func}()\{\cdots\}); \mathtt{deref}\ \ l) \to^*$$
$$(\sigma', (v'; \mathtt{deref}\ \ l)) \to^* (\sigma', \sigma'(l))$$

Then $v = \sigma'(l)$. Now define a world w that has one island ι_l^{std} for every location $l \in \Sigma$:

$$\mathrm{dom}(w) = \{1..|\Sigma|\} \wedge w(\{1..|\Sigma|\}) = \{\iota_l^{std} \mid l \in \Sigma\}$$

Take i the number of steps in the middle part of the above evaluation and $n > i$. It is easy to check that $(n, w') \in [\![\Sigma]\!]_{IO^{std},\mathit{Ref}^{std}}$ for all $w' \sqsupseteq w$ and effect parametricity then gives us that $(n, \mathit{attacker}) \in \mathcal{E}[IO^{std}\ \ \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w'$.

The following island captures the intended usage protocol on l:

$$\iota_{tkt,l,k} \overset{\mathsf{def}}{=} ((l,k), \sqsubseteq_{tkt}, \sqsubseteq_{tkt}, H_{tkt}) \qquad \text{for } k \text{ even}$$
$$(l,k) \sqsubseteq_{tkt} (l', k') \text{ iff } l = l' \wedge k' \geq k \wedge k', k \text{ even}$$
$$H_{tkt}\ \ (l,k)\ w \overset{\mathsf{def}}{=} \{(n, \{l \mapsto k\}) \mid n \in \mathbb{N}\}$$

Take $j \notin \mathrm{dom}(w)$ and call $w' = w[j \mapsto \iota_{tkt,l,0}]$. Then clearly $w' \sqsupseteq w$.

We will show that

$$(n, \mathtt{func}()\{\mathtt{return}\ \ (\mathtt{let}(v = \mathtt{deref}\ \ l)\{l := v + 2; v\})\}) \in$$
$$([\mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}] \to IO^{std}\ \ \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}})\ w' \subseteq$$
$$\mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'$$

So, for $w'' \sqsupseteq w'$ and $i < n$, we need to show that $(i, \mathtt{let}(v = \mathtt{deref}\ \ l)\{l := v + 2; v\})$ is in $\mathcal{E}[IO^{std}\ \ \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w''$ or, sufficiently, in $IO^{std}\ \ \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w''$. The expression is clearly not a value so it remains to prove for $0 < i' \leq i$ and $\sigma'' :_i w''$ that $(\sigma'', \mathtt{let}(v = \mathtt{deref}\ \ l)\{l := v + 2; v\}) \to^{i'} (\sigma''', e')$, with $e' \in \mathit{Val}$ implies that there exists a $w''' \sqsupseteq^{\mathrm{pub}} w''$ with $\sigma''' :_{i-i'} w'''$ and $(i - i', e') \in \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'''$. But the evaluation and $\sigma'' :_i w''$ together imply that $\sigma''' = \sigma''[l \mapsto \sigma''(l) + 2]$, $e' = \sigma''(l)$ and $\sigma''(l)$ is an even number value. Then clearly $(i - i', e') \in \mathit{Cnst} \subseteq \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'''$. Finally, $\sigma''' :_{i-i'} w'''$ follows by monotonicity and uniformity for $w''' = w''[j \mapsto \iota_{tkt,l,\sigma(l)+2}] \sqsupseteq^{\mathrm{pub}} w''$.

We then have $\sigma[l \mapsto 0] :_n w'$ and we can combine Lemma 65, the sub-evaluation $(\sigma[l \mapsto 0], \mathit{attacker}\ (\mathtt{func}()\{\cdots\})) \to^* (\sigma', v')$, the fact that

$$(n, \mathit{attacker}) \in \mathcal{E}[IO^{std}\ \ \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w',$$

the compatibility lemma for applications (Lemma 59) and

$$(n, \mathtt{func}()\{\mathtt{return}\ \ (\mathtt{let}(v = \mathtt{deref}\ \ l)\{l := v + 2; v\})\}) \in$$
$$\mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'$$

to obtain a $w'' \sqsupseteq^{\mathrm{pub}} w'$ such that $\sigma' :_{n-i} w''$ and $(n - i, v') \in \mathtt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w''$. From the former, it is easy to deduce that $\sigma'(l)$ is an even number value ≥ 0. □

$$\iota^{\mathrm{dom}}_{l,tree,P} \stackrel{\mathsf{def}}{=} ((l, tree), \sqsubseteq^{\mathrm{dom}}, \sqsubseteq^{\mathrm{dom}}, H^{\mathrm{dom}}_{P})$$

$$H^{\mathrm{dom}}_{P}\ (l, tree)\ w \stackrel{\mathsf{def}}{=}$$

$$\left\{ (n, h) \;\middle|\; \begin{array}{l} h = (l \mapsto v) \wedge n = 0 \text{ or } (n-1, v) \in \\ \qquad P \cdot (roll^{-1}_{\hat{W}}\ w) \text{ if } tree = v \wedge \\ h = h_1 \uplus \cdots \uplus h_n \uplus h' \wedge \\ \qquad h' \text{ represents list } ((id_1, l_1), \cdots, (id_n, l_n)) \text{ at } l \wedge \\ (n-1, h_i) \in H^{\mathrm{dom}}_{P'_i}\ (l_i, t_i)\ w \text{ for all } i = 1..n \text{ and } P'_i\ \overline{id}' = P\ (id_i : \overline{id}') \\ \hfill \text{if } tree = (id_1 \mapsto t_1, \cdots, id_n \mapsto t_n) \end{array} \right\}$$

$$(l', tree') \sqsupseteq^{\mathrm{dom}} (l, tree) \text{ iff } l = l'$$

$$(l_1, t_1) \sqsupseteq^{\mathrm{r-dom}}_{p} (l_2, t_2) \text{ iff}$$
$$l_1 = l_2 \wedge (\forall t'_1, t_f.\ t_1 = t_f[p \mapsto t'_1] \Rightarrow \exists t'_2.\ t_2 = t_f[p \mapsto t'_2])$$
$$\iota^{\mathrm{r-dom}}_{l,t,p,P} \stackrel{\mathsf{def}}{=} ((l, t), \sqsubseteq^{\mathrm{dom}}, \sqsubseteq^{\mathrm{r-dom}}_{p}, H^{\mathrm{dom}}_{P})$$

Lemma 2. *Assume that $document$'s methods $getChild$, $parent$, $getProp$, $setProp$, $addChild$ and $delChild$ behave in the "obvious" way (to be defined formally in the TR) in stores that contain the representation of a state of the the DOM. If $w(j) = \iota^{\mathrm{dom}}_{l,t,P}$ for some j, t and P, and for n arbitrarily large, we have that $\sigma :_n w$, and ad is a closed expression and $(\sigma, initWebPage(document, ad)) \to^i (\sigma', v)$ for $i \leq n$, then $v = \mathtt{true}$.*

Proof. For reference, we repeat the definitions of $rnode$ and $initWebPage$:

$$rnode \stackrel{\mathsf{def}}{=} \mathtt{func}(node, d)$$
$$\left\{ \begin{array}{r} getChild = \mathtt{func}(id) \left\{ rnode(node.getChild(id), d+1) \right\} \\ parent = \mathtt{func}() \left\{ \begin{array}{l} \mathtt{if}\ (d \leq 0)\ \{\mathtt{error}\}\ \mathtt{else} \\ \quad \{rnode(node.parent(), d-1)\} \end{array} \right\} \\ getProp = \mathtt{func}(id)\{node.getProp(id)\} \\ setProp = \mathtt{func}(id, v)\{node.setProp(id, v)\} \\ addChild = \mathtt{func}(id) \left\{ rnode(node.addChild(id), d+1) \right\} \\ delChild = \mathtt{func}(id)\{node.delChild(id)\} \end{array} \right\}$$

$$initWebPage \stackrel{\mathsf{def}}{=} \mathtt{func}(document, ad)$$
$$\left\{ \begin{array}{l} document.setProp(\text{“}someProperty\text{”}, 42) \\ \mathtt{let}\ (adNode = document.addChild(\text{“}ad\_div\text{”})) \\ \mathtt{let}\ (rAdNode = rnode(adNode, 0)) \\ ad.initialize(rAdNode) \\ document.getProp(\text{“}someProperty\text{”}) == 42 \end{array} \right\}$$

First, we list the precise assumptions on $document$, i.e. we specify what we mean with $document$'s methods behaving in the "obvious" way. We define a predicate (not necessarily uniform or monotonous) for specifying the behaviour of a DOM operation. Given a reference l (identifying the DOM root node) and a function

$$Post \in Tree \to (Tree \times (W \to_{ne} Pred(Val))$$

we define $DOMOp_{l,P,Post}$ as follows:

$$DOMOp_{l,P,Post} : W \to_{ne} Pred(Cmd))$$
$$DOMOp_{l,P,Post}\ w \stackrel{\mathsf{def}}{=}$$
$$\left\{ (n, e) \;\middle|\; \begin{array}{l} e \notin Val \wedge \text{for all } \sigma_r, \sigma_f, \sigma', t, 0 < i \leq n, e'. \\ \quad (n, \sigma_r) \in H^{\mathrm{dom}}_{P}\ (l, t)\ w \wedge Post(t) = (t', R) \wedge \\ \quad (\sigma_r \uplus \sigma_f, e) \to^i (\sigma', e') \wedge e' \in Val \Rightarrow \\ \exists \sigma'_r.\ \sigma' = \sigma'_r \uplus \sigma_f \wedge (n-i, \sigma'_r) \in H^{\mathrm{dom}}_{P}\ (l, t')\ w \wedge \\ \hfill (n-i, e') \in R\ w \end{array} \right\}$$

The postcondition $Post$ specifies, given an input state of the DOM tree, the output tree of the DOM and a predicate identifying valid result values in terms of the predicate P identifying valid values for DOM properties at certain paths and a current world. Note that when $Post(t) = (\_, Empty)$, this implies that the expression must not evaluate to a value, i.e. it must get stuck or its step-index must run out. The predicate $DOMOp_{l,P,Post}$ n-accepts expressions e which behave according to the postcondition $Post$: when e is evaluated to a value in a store with a part σ_r that satisfies the H_P^{dom} predicate for reference l, a DOM state t and a properties predicate P, the resulting store should satisfy the H_P^{dom} predicate for the postcondition-specified new state of the heap and the result value should satisfy the appropriate predicate.

We also define

$$Undef : W \to_{mon,ne} UPred(Val)$$

$$Undef\ w \overset{\mathsf{def}}{=} \mathbb{N} \times \{\mathtt{undef}\}$$

$$Empty : W \to_{mon,ne} UPred(Val)$$

$$Empty\ w \overset{\mathsf{def}}{=} \emptyset$$

For simplicity, we let erroneous calls to DOM methods get stuck: using child or property id's that are not strings, invoking $parent()$ on the root node, reading a property that does not exist, adding a child under a name that is already taken, deleting or getting a child that doesn't exist or generally working with a node that represents a node that is no longer part of the DOM tree. This avoids having to do a lot of checking of result values of DOM functions in our examples.

We say that $(n, node) \in NodeSpec_{l,p}$ if $node \in Val$ and for any $P \in Path \to W \to_{mon,ne} UPred(Val)$, we have that:

- For any w, $i \leq n$ and $id \in Val$, we have that

  $$(i, node.getChild(id)) \in \mathcal{E}[DOMOp_{l,P,PostGetChild_{p,id}}]\ w$$

  with

  $$PostGetChild_{p,id}(t) \overset{\mathsf{def}}{=} \begin{cases} (t, NodeSpec_{l,(id:p)}) & \text{if } id \text{ is a string constant } \wedge \exists t'', t'''.\ t = t''[(id:p) \mapsto t'''] \wedge t''' \text{ not a leaf} \\ (t, Empty) & \text{otherwise} \end{cases}$$

- For any w, $i \leq n$, we have that

  $$(i, node.parent()) \in \mathcal{E}[DOMOp_{l,P,PostParent_p}]\ w$$

  with

  $$PostParent_p(t) \overset{\mathsf{def}}{=} \text{ iff } t = t' \wedge \begin{cases} (t, NodeSpec_{l;p'}) & \text{if } p = (p', \_) \\ (t, Empty) & \text{otherwise} \end{cases}$$

- For any w, $i \leq n$, $id \in Val$, we have that

  $$(i, node.getProp(id)) \in \mathcal{E}[DOMOp_{l,P,PostGetProp_{p,id}}]\ w$$

  with

  $$PostGetProp_{p,id}(t) \overset{\mathsf{def}}{=} \begin{cases} (t, P\ (p, id)) & \text{if } \begin{array}{l} id \text{ is a string constant } \wedge \\ \exists t'', t''', v'.\ t = t''[(p, id) \mapsto v'] \end{array} \\ (t, Empty) & \text{otherwise} \end{cases}$$

- For any w, $i \leq n$, value id, $(i, v) \in P\ (p, id)\ w$, we have that

  $$(i, node.setProp(id, v)) \in \mathcal{E}[DOMOp_{l,P,PostSetProp_{p,id,v}}]\ w$$

  with

  $$PostSetProp_{p,id,v}(t) \overset{\mathsf{def}}{=} \begin{cases} (t'', \lambda w.\ \mathbb{N} \times \{v\}) \text{ with } t'' = t'[(p:id) \mapsto v'] & \text{if } \begin{array}{l} id \text{ is a string constant } \wedge \\ \exists t', v''.\ t = t'[(p:id) \mapsto v''] \end{array} \\ (t'', \lambda w.\ \mathbb{N} \times \{v\}) \text{ with } t'' = t'[p \mapsto (\overline{id' \mapsto tree', id \mapsto v'})] & \text{if } \begin{cases} id \text{ is a string constant } \wedge \\ \exists t', \overline{id'}, \overline{tree'}.\ id \notin \{id\} \wedge \\ t = t'[p \mapsto (\overline{id' \mapsto tree'})] \end{cases} \\ (t, Empty) & \text{otherwise} \end{cases}$$

- For any w, $i \leq n$, value id, we have that

  $$(i, node.addChild(id)) \in \mathcal{E}[DOMOp_{l,P,PostAddChild_{p,id}}]\ w$$

with

$$\mathit{PostAddChild}_{p,id}(t) \overset{\mathsf{def}}{=} \begin{cases} (t'', \lambda w.\ \mathit{NodeSpec}_{l,\overline{(id:p)}}) \text{ with} & id \text{ is a string constant } \wedge \\ \quad t'' = t'[p \mapsto (\overline{id' \mapsto tree'}, id \mapsto ())] & \text{if} \quad \exists t', \overline{id'}, \overline{tree'}.\ id \notin \{\overline{id'}\} \wedge \\ & \quad t = t'[p \mapsto (\overline{id' \mapsto tree'})] \\ (t, \mathit{Empty}) & \text{otherwise} \end{cases}$$

- For any w, $i \leq n$, string constant id, we have that

  $$(i, \mathit{node.delChild}(id)) \in \mathit{DOMOp}_{l,P,\mathit{PostDelChild}_{p,id}}\ w$$

  with

$$\mathit{PostDelChild}_{p,id}(t) \overset{\mathsf{def}}{=} \begin{cases} & id \text{ is a string constant } \wedge \\ (t'', \mathit{Undef}) \text{ with } t'' = t'[p \mapsto (\overline{id' \mapsto tree'})] & \text{if} \quad \exists t', \overline{id'}, \overline{tree'}, \mathit{child}.\ id \notin \{\overline{id'}\} \wedge \\ & \quad t = t'[p \mapsto (\overline{id' \mapsto tree'}, id \mapsto \mathit{child})] \\ (t, \mathit{Empty}) & \text{otherwise} \end{cases}$$

The definition of $\mathit{NodeSpec}$ is recursive, so we need to define it as another fixpoint of a contractive function. We omit details but this can be done correctly, because DOMOp is contractive in its Post argument.

We require that for some reference l, and for some w, $P \in W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Val})$ and t, we have that $(n, \mathit{document}) \in \mathit{NodeSpec}_{l,\cdot}$ with $\cdot$ the empty path and l the reference for which we assumed $w(j) = \iota^{\mathrm{dom}}_{l,t,P}$.

So now take the evaluation $(\sigma, \mathit{initWebPage}(\mathit{document}, \mathit{ad})) \rightarrow^{i} (\sigma', v)$. It must factor as

$$\begin{aligned} &(\sigma, \mathit{initWebPage}(\mathit{document}, \mathit{ad})) \rightarrow^{i_1} \\ &(\sigma_1, e_1) \rightarrow^{i_2} (\sigma_2, \mathit{ad.initialize}(\mathit{rAdNode}'); \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42) \rightarrow^{i_3} \\ &\qquad\qquad (\sigma_3, \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42) \rightarrow^{i_4} (\sigma', v' == 42) \rightarrow (\sigma', v) \end{aligned}$$

with

$$e_1 \overset{\mathsf{def}}{=} \begin{array}{l} \texttt{let}\ (\mathit{rAdNode} = \mathit{rnode}(\mathit{adNode}', 0)) \\ \mathit{ad.initialize}(\mathit{rAdNode}) \\ \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42 \end{array}$$

for some value adNode' and $(\sigma_1, \mathit{rnode}(\mathit{adNode}, 0)) \rightarrow^{i_2} (\sigma_2, \mathit{rAdNode}')$.

We know that $\sigma = \sigma_r \uplus \sigma_f$ with $(n, \sigma_r) \in H^{\mathrm{dom}}_P\ (l,t)\ w$, and so the contract of $\mathit{document}$ tells us that $\sigma_1 = \sigma_{r,1} \uplus \sigma_f$ with $(n - i_1, \sigma_{r,1}) \in H^{\mathrm{dom}}_P\ (l, t_1)\ w$, $(n_1, \mathit{adNode}') \in \mathit{NodeSpec}_{j,\text{“}\mathit{ad\_div}\text{”}}$ and $t_1 = t'[\text{“}\mathit{ad\_div}\text{”} \mapsto ()][\text{“}\mathit{someProperty}\text{”} \mapsto 42]$ for some t'. Define $n_1 = n - i_1$ and $w_1 \overset{\mathsf{def}}{=} w[j \mapsto \iota^{\mathrm{dom}}_{l,t_1,P}]$. Then $w_1 \sqsupseteq w$ and $\sigma_1 :_{n_1} w_1$.

Now define path $p = \text{“}\mathit{ad\_div}\text{”}$ and define $P^{\mathrm{r-dom}}_p$ as follows:

$$P^{\mathrm{r-dom}}_p\ p'\ w'' \overset{\mathsf{def}}{=} \begin{cases} \texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w'' & \text{if } p \text{ is a prefix of } p' \\ P\ w_1 & \text{otherwise} \end{cases}$$

and define w'_1 as the single island world $j \mapsto \iota^{\mathrm{r-dom}}_{l,t_1,p,P^{\mathrm{r-dom}}_p}$. The intuition here is that $P^{\mathrm{r-dom}}_p$ defines a new invariant on properties of the DOM, which requires that properties under path p must be valid in a world with the limited authority modelled by public transition relation $\sqsupseteq^{\mathrm{r-dom}}_p$ but properties at other paths must only be valid in the world w_1 with full authority on the DOM. Those other properties must not even preserve the invariant on properties under p. Note that w'_1 is *not* a future world of w_1.

We now prove that $(n_1, \mathit{rnode}(\mathit{adNode}', 0))$ is in $\mathcal{E}[\texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w'_1$.

Proof. First, we generalise and prove that if $(n'_1, \mathit{elt}) \in \mathit{NodeSpec}_{l,p'}$ with $p' = p\ p''$ and $\mathrm{len}(p'') = d$ then $(n'_1, \mathit{rnode}(\mathit{elt}, d)) \in \mathcal{E}[\texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w''_1$ in any $w''_1 \sqsupseteq w'_1$ and for any $n'_1 \leq n_1$ and we prove this by induction on n'_1.

We easily have that $\mathit{rnode}(\mathit{elt}, d)$ purely evaluates to a record value in one step, so it suffices to prove that this value $n'_1 - 1$-satisfies $\texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w''_1$. For this, it suffices to prove that all fields of the resulting record value $n'_1 - 2$-satisfy $([\texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}] \rightarrow IO^{std}\ \texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}})\ w''_1$. So, take an arbitrary $w'' \sqsupseteq w''_1$ and $i < n'_1 - 2$, then we need to prove that

- *getChild*: for $(i, id) \in \texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}\ w''$, we need to show that $(i, \mathit{rnode}(\mathit{elt.getChild}(id), d+1))$ is in $\mathcal{E}[IO^{std}\ \texttt{JSVal}_{IO^{std},\mathit{Ref}^{std}}]\ w''$. So, assume that $i' \leq i$ and $\mathit{rnode}(\mathit{elt.getChild}(id), d+1) \rightarrow^{i'} e'$ and e' is a command and for $0 < i'' < i - i'$, $\sigma_r :_{i-i'} w''$, $(\sigma_r \uplus \sigma_f, e') \rightarrow^{i''} (\sigma', e'')$ and $e'' \in \mathit{Val}$. We know that $(i, \mathit{elt.getChild}(id)) \in \mathcal{E}[\mathit{DOMOp}_{l,P,\mathit{PostGetChild}_{p,id}}]\ w''$ from the fact that $(n'_1, \mathit{elt}) \in \mathit{NodeSpec}_{l,p'}$. By Lemma 32, and the fact that

DOMOp excludes values, we know that $\mathit{elt.getChild}(\mathit{id}) \rightarrow^{i'} e'''$ and $e' = \mathit{rnode}(e''', d+1)$ and $(i - i', e''') \in \mathit{DOMOp}_{l,P,\mathit{PostGetChild}_{p,id}}\ w''$. By Lemmas 33 and 29, we get a sub-evaluation $(\sigma_r \uplus \sigma_f, e''') \rightarrow^{i'''} (\sigma'', e'''')$ with $e'''' \in \mathit{Val}$. From $w'' \sqsupseteq w_1'' \sqsupseteq w_1'$, we get that $w''(j) = ((l,t'), \sqsubseteq^{\mathrm{dom}}, \sqsubseteq', H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}})$ for some t' and for $\sqsubseteq' \subseteq \sqsubseteq_p^{\mathrm{r-dom}}$. From $\sigma_r :_{i-i'} w''$, there must then be a σ_r' such that $\sigma_r = \sigma_r' \uplus \sigma_f'$ and $(i - i', \sigma_r') \in H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}}\ (l, t')\ w''$, so that we can apply the definition of DOMOp to obtain that $\sigma'' = \sigma_r'' \uplus \sigma_f' \uplus \sigma_f$ and $(i - i' - i''', \sigma_r'') \in H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}}\ (l, t'')\ w''$ for $t'' = t'$ and $(i - i' - i''', e'''') \in \mathit{NodeSpec}_{l,id:p}$. The remaining evaluation $(\sigma'', \mathit{rnode}(e'''', d+1) \rightarrow^{i''-i'''} (\sigma', e'')$ is easily seen to be pure (so that $\sigma' = \sigma''$) and together with the induction hypothesis, we get that $(i - i' - i'', e'') \in \mathtt{JSVal}_{IO^{std}, \mathit{Ref}^{std}}\ w_1''$. We can also deduce that $\sigma' = (\sigma_r'' \uplus \sigma_f') \uplus \sigma_f$ with $\sigma_r'' \uplus \sigma_f' :_{i-i'-i''} w''$ by uniformity.

- The other cases are similar.

□

We also have that $(n_1, \sigma_{r,1}) \in H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}}\ (l, t_1)\ w_1'$. This follows from the fact that $(n_1, \sigma_{r,1}) \in H^{\mathrm{dom}}_{P}\ (l, t_1)\ w$, and the fact that $H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}}\ (l, t_1)\ w_1'$ is equal to $H^{\mathrm{dom}}_{P}\ (l, t_1)\ w_1$ because we know that $t_1 = t'[\text{“}\mathit{ad\_div}\text{”} \mapsto ()][\text{“}\mathit{someProperty}\text{”} \mapsto 42]$ for some t'. This is equivalent to $\sigma_{r,1} :_{n_1} w_1'$.

The above result about rnode tells us that $(n_1, \mathit{rnode}(\mathit{adNode}', 0)) \in \mathcal{E}[\mathtt{JSVal}_{IO^{std}, \mathit{Ref}^{std}}]\ w_1'$. By the evaluation $(\sigma_1, \mathit{rnode}(\mathit{adNode}, 0)) \rightarrow^{i_2} (\sigma_2, \mathit{rAdNode}')$ and by inspection of rnode we get that the evaluation must be pure, $\sigma_2 = \sigma_1$ and by definition of $\mathcal{E}$, $(n_1 - 1, \mathit{rAdNode}') \in \mathtt{JSVal}_{IO^{std}, \mathit{Ref}^{std}}\ w_1'$. We call $n_2 \overset{\mathsf{def}}{=} n_1 - 1$.

Since ad is closed, the Fundamental Theorem tells us that $(n_2, \mathit{ad.initialize}) \in \mathcal{E}[IO^{std}\ \mathtt{JSVal}_{IO^{std}, \mathit{Ref}^{std}}]\ w_1'$, so by Lemma 59, we have that $(n_2, \mathit{ad.initialize}(\mathit{rAdNode})) \in \mathcal{E}[IO^{std}\ \mathtt{JSVal}_{IO^{std}, \mathit{Ref}^{std}}]\ w_1'$.

The evaluation

$$(\sigma_2, \mathit{ad.initialize}(\mathit{rAdNode}'); \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42) \rightarrow^{i_3}$$
$$(\sigma_3, \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42)$$

must have a sub-evaluation

$$(\sigma_2, \mathit{ad.initialize}(\mathit{rAdNode}')) \rightarrow^{i_3 - 1} (\sigma_3, v'')$$

for which Lemma 65 now produces a $w_3 \sqsupseteq^{\mathrm{pub}} w_1'$ such that $\sigma_3 = \sigma_{r,3} \uplus \sigma_f$ with $\sigma_{r,3} :_{n_2 - i_3 + 1} w_3$.

From $w_3 \sqsupseteq^{\mathrm{pub}} w_1'$, it follows that $w_3(j) = \iota^{\mathrm{r-dom}}_{l,t',p,P_p^{\mathrm{r-dom}}}$ with $(l, t') \sqsupseteq_p^{\mathrm{r-dom}} (l, t_1)$. From this, we can deduce that $t' = t''[\text{“}\mathit{someProperty}\text{”} \mapsto 42]$ for some t''. We also have that $\sigma_{r,3} :_{n_3} w_3$ for $n_3 \overset{\mathsf{def}}{=} n_2 - i_3 + 1$ which implies that $(n_3, \sigma_{3,r}) \in H^{\mathrm{dom}}_{P_p^{\mathrm{r-dom}}}\ (l, t')\ w_3$. The evaluation $(\sigma_3, \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”}) == 42) \rightarrow^{i_4} (\sigma', v' == 42)$ must have a sub-evaluation $(\sigma_3, \mathit{document.getProp}(\text{“}\mathit{someProperty}\text{”})) \rightarrow^{i_4} (\sigma', v')$ so that the spec of $\mathit{document}$ implies that $v' = 42$ and we can conclude from the remaining evaluation that $v = \mathtt{true}$. □

Appendix G. Proofs and details for Section 5

First some further definitions. Given a store σ and expression e, Maffeis et al. define the *trace* $\tau((\sigma, e))$ as the possibly infinite sequence of states (σ_i', e_i') such that $(\sigma, e) \rightarrow (\sigma_1', e_1') \rightarrow (\sigma_2', e_2') \rightarrow \cdots$. The set of store-affecting actions $\mathtt{act}(\sigma)$ is $\mathrm{dom}(\sigma) \times \mathbb{D}$. We also note that our use of action-labeled evaluation judgement is an alternative to Maffeis et al.'s acc function: $\mathrm{acc}(\sigma, e) \overset{\mathsf{def}}{=} A$ iff $(\sigma, e) \rightarrow_A (\sigma', e')$ for some σ' and e'.

Lemma 67. *λ_{JS} satisfies the properties* RG-AUTH1, RG-AUTH2, RG-CONN *and* RG-NOAMPL, *formulated in Section 5. We list them here again for reference:*

$(\sigma, e) \rightarrow_A (\sigma', e')$ implies that

- RG-AUTH1*:* $A \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$
- RG-AUTH2*:* $\mathrm{auth}(\sigma', e') \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$

Furthermore, if $(\sigma, e) \rightarrow_A^* (\sigma', v) \not\rightarrow$ and $v \in \mathit{Val}$, then for any location l, we must have:

- RG-CONN*:* $A \not\triangleright \mathrm{cAuth}(\sigma, l)$ *implies that* $\mathrm{cAuth}(\sigma', l) = \mathrm{cAuth}(\sigma, l) \cup \{(l, \mathtt{r}), (l, \mathtt{w})\}$
- RG-NOAMPL*:* $A \triangleright \mathrm{cAuth}(\sigma, l)$ *implies that*

$$\mathrm{cAuth}(\sigma', l) \subseteq \mathrm{cAuth}(\sigma, l) \cup$$
$$\{(l, \mathtt{r}), (l, \mathtt{w})\} \cup \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$$

Proof. Remember that $\mathrm{cAuth}(\sigma, l)$ is defined as the least set of actions CA such that $\{(l, \mathtt{r}), (l, \mathtt{w})\} \subseteq CA$ and

$$\left(\bigcup_{(l,\mathtt{r})\in A} \mathrm{tCap}(h(l)) \times \mathbb{D}\right) \subseteq CA$$

and that $\mathrm{auth}(\sigma, e)$ is defined as $\bigcup_{c\in \mathrm{tCap}(e)} \mathrm{cAuth}(\sigma, c)$.

Assume that $(\sigma, e) \to_A (\sigma', e')$. RG-AUTH1 easily follows by case analysis of the impure evaluation judgement. RG-AUTH2 follows using an easy lemma that $e_1 \to e_2$ implies $\mathrm{tCap}(e_1) \supseteq \mathrm{tCap}(e_2)$.

Then assume that $(\sigma, e) \to^* (\sigma', v) \not\to$ and $v \in \mathit{Val}$. If $A \not\triangleright \mathrm{cAuth}(\sigma, l)$, then it easily follows that for all l with $(l, \mathtt{r}) \in \mathrm{cAuth}(\sigma, l)$, $\sigma'(l) = h(l)$. The set $CA = \mathrm{cAuth}(\sigma, l)$ then still satisfies the defining requirements for $\mathrm{cAuth}(\sigma', l)$: $\{(l, \mathtt{r}), (l, \mathtt{w})\} \subseteq CA$ and

$$\left(\bigcup_{(l,\mathtt{r})\in CA} \mathrm{tCap}(\sigma'(l)) \times \mathbb{D}\right) \subseteq CA$$

This is because for every $(l, \mathtt{r}) \in CA$, we know that $\sigma'(l) = h(l)$, and that $\mathrm{tCap}(\sigma'(l)) \times \mathbb{D} = \mathrm{tCap}(h(l)) \times \mathbb{D} \subseteq CA$.

If $A \triangleright \mathrm{cAuth}(\sigma, l)$, then we define $CA' \overset{\mathsf{def}}{=} \mathrm{cAuth}(\sigma, l) \cup \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$ and we prove that it satisfies the requirements for $\mathrm{cAuth}(\sigma', l)$. Clearly, $\{(l, \mathtt{r}), (l, \mathtt{w})\} \subseteq \mathrm{cAuth}(\sigma, l) \subseteq CA'$. We prove that

$$\left(\bigcup_{(l,\mathtt{r})\in CA'} \mathrm{tCap}(\sigma'(l)) \times \mathbb{D}\right) \subseteq CA'$$

for an arbitrary expression e' (not necessarily a value) by induction on the length of the evaluation $(\sigma, e) \to_A^* (\sigma', v)$. For a zero-length evaluation, $h = \sigma'$ and if $(l', \mathtt{r}) \in CA'$, then either

- $(l', \mathtt{r}) \in \mathrm{cAuth}(\sigma, l)$, in which case $\mathrm{tCap}(h(l')) \times \mathbb{D} \subseteq \mathrm{cAuth}(\sigma, l) \subseteq CA'$.
- $(l', \mathtt{r}) \in \mathrm{auth}(\sigma, e)$. Then, $(l', \mathtt{r}) \in \mathrm{cAuth}(\sigma, l'')$ for some $l'' \in \mathrm{tCap}(e)$. But then also $\mathrm{tCap}(h(l')) \times \mathbb{D} \subseteq \mathrm{cAuth}(\sigma, l'') \subseteq \mathrm{auth}(\sigma, e) \subseteq CA'$.
- $(l', \mathtt{r}) \in \mathrm{nauth}(\sigma', \sigma) = \emptyset$: not possible.

Now suppose that $(\sigma, e) \to^* (\sigma', e') \to (\sigma'', e'')$. Then if we know that $CA' = \mathrm{cAuth}(\sigma, l) \cup \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$ satisfies the defining conditions for $\mathrm{cAuth}(\sigma', l)$, we prove that $CA'' = \mathrm{cAuth}(\sigma, l) \cup \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma)$ satisfies the requirements for $\mathrm{cAuth}(\sigma'', l)$. It then follows that it must be an upper bound for $\mathrm{cAuth}(\sigma'', l)$, because that set was defined as the least set satisfying the requirements. It's clear that $\{(l, \mathtt{r}), (l, \mathtt{w})\} \subseteq \mathrm{cAuth}(\sigma, l) \subseteq CA''$. So, take $(l', \mathtt{r}) \in CA''$. We need to prove that $\mathrm{tCap}(\sigma''(l')) \times \mathbb{D} \subseteq CA''$. Now either

- $l' \in \mathrm{nauth}(\sigma'', \sigma')$. Then the step $(\sigma', e') \to (\sigma'', e'')$ must be an application of evaluation rule E-REF, i.e. $e' = E\langle \mathtt{ref}\ v\rangle$, $e'' = E\langle l'\rangle$, $\sigma'' = \sigma'[l' \mapsto v]$. Then $\sigma''(l') = v$ and $\mathrm{tCap}(v) \times \mathbb{D} \subseteq \mathrm{tCap}(e') \times \mathbb{D} \subseteq \mathrm{auth}(\sigma', e')$. A repeated application of property RG-AUTH2 above tells us that $\mathrm{auth}(\sigma', e') \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma) \subseteq A' \subseteq A''$.
- $l' \in \mathrm{dom}(\sigma'') \cap \mathrm{dom}(\sigma')$ and $\sigma''(l') \neq \sigma'(l')$. Then the step $(\sigma', e') \to (\sigma'', e'')$ must be an application of evaluation rule E-SETREF with $e' = E\langle l' = v\rangle$, $e'' = E\langle v\rangle$ and $\sigma'' = \sigma'[l' \mapsto v]$. Then $\sigma''(l') = v$ and $\mathrm{tCap}(v) \times \mathbb{D} \subseteq \mathrm{tCap}(e') \times \mathbb{D} \subseteq \mathrm{auth}(\sigma', e')$. Again by a repeated application of property RG-AUTH2 above, we get $\mathrm{auth}(\sigma', e') \subseteq \mathrm{auth}(\sigma, e) \cup \mathrm{nauth}(\sigma', \sigma) \subseteq A' \subseteq A''$.
- $l' \in \mathrm{dom}(\sigma'') \cap \mathrm{dom}(\sigma')$ and $\sigma''(l') = \sigma'(l')$. Then $(l', \mathtt{r})$ is also in A' and $\mathrm{tCap}(\sigma''(l')) \times \mathbb{D} = \mathrm{tCap}(\sigma'(l)) \times \mathbb{D} \subseteq A' \subseteq A''$.

□

G.1. More details about `deepInspect`

Here are a bit more details about the argument involving the `deepInspect` primitive mentioned in Section 5.2.

Consider a hypothetical language λ_{JS}^{di} where indirect references can be converted into direct references using a `deepInspect` primitive, making the private instance state of objects public. The formal semantics could look like this with $\mathit{internals}(v) \overset{\mathsf{def}}{=} \{l \mid l \sqsubseteq v\}$:

$$\frac{\mathit{internals}(v) = \{l_1 \cdots l_n\} \qquad 1 \le k \le n}{\mathtt{deepInspect}_k(v) \hookrightarrow l_k} \qquad \text{(E-INSPECT)}$$

Such a primitive contradicts basic principles of the object capability model, and breaks most examples relying on the model (particularly all examples in Section 4). For example, we could use `deepInspect` on the function $\mathit{dispTkt}$ to obtain a direct reference to o. Nevertheless, λ_{JS}^{di} still satisfies the reference graph dynamics properties and the topology-only bound on authority. Clearly, our Fundamental Theorem is strictly stronger than the standard reference graph dynamics properties.

It is easy to check that the above proof of the reference graph properties is unaffected by the addition of `deepInspect` and E-DEEPINSPECT.

Appendix H. Proofs and details for Section 6

Lemma 68. *The effect interpretation* $(\mathit{IO}^{rgn}_{j}, \mathit{Ref}^{rgn}_{j})$ *is well-defined and the island* $\iota^{rgn}_{j,L}$ *is well-defined.*

Proof. IO^{rgn}_{j}, Ref^{rgn}_{j} and H^{rgn}_{j} are well-defined. We construct them jointly as the Banach fixpoint of contractive function $\mathit{RefIO}^{rgn}_{j,rec}$. In this proof, we write EffInt for the set

$$(W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Loc})) \times ((W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Val})) \rightarrow_{ne} (W \rightarrow_{ne} \mathit{Pred}(\mathit{Cmd})))$$

$$\mathit{Ref}^{rgn}_{j,rec} : \mathit{EffInt} \rightarrow_{ne} W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Loc})$$

$$\mathit{Ref}^{rgn}_{j,rec}\ (\rho,\mu)\ w \overset{\mathsf{def}}{=} \left\{ (n,l) \;\middle|\; \begin{array}{r} \exists L \ni l.\ w(j) = (L, \subseteq, \subseteq, H) \wedge \\ H =_{n+1} (H^{rgn}_{rec}\ (\rho,\mu)) \end{array} \right\}$$

$$\mathit{IO}^{rgn}_{j,rec} : \mathit{EffInt} \rightarrow_{ne} (W \rightarrow_{mon,ne} \mathit{UPred}(\mathit{Val})) \rightarrow_{ne} (W \rightarrow_{ne} \mathit{Pred}(\mathit{Cmd}))$$

$$\mathit{IO}^{rgn}_{j,rec}\ (\rho,\mu)\ P\ w \overset{\mathsf{def}}{=} \left\{ (n,e) \;\middle|\; \begin{array}{l} (e \in \mathit{Val} \Rightarrow (n,e) \in P\ w) \wedge \\ \forall L, H.\ \text{ if } w(j) = (L, \subseteq, \subseteq, H) \wedge H =_{n+1} (H^{rgn}_{rec}\ (\rho,\mu)) \\ \quad \text{then } \forall 0 < i \leq n, \sigma_r :_n w, \sigma_f. (\sigma_r \uplus \sigma_f, e) \rightarrow^{i}_{A} (\sigma', e') \Rightarrow \\ \qquad \exists \sigma'_r, w' \sqsupseteq^{\mathrm{pub}} w.\ \sigma' = \sigma'_r \uplus \sigma_f \wedge \\ \qquad A \subseteq ((L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbf{D}) \wedge \\ \qquad w'(j) = (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H) \wedge \\ \qquad (n-i, e') \in \mathcal{E}[\mu\ P]\ w' \wedge \sigma'_r :_{n-i} w' \end{array} \right\}$$

$$H^{rgn}_{rec} : \mathit{EffInt} \rightarrow_{ne} \mathcal{P}(\mathit{Loc}) \rightarrow \hat{W} \rightarrow_{mon,ne} \mathit{UPred}(\mathrm{Store})$$

$$H^{rgn}_{rec}\ (\rho,\mu)\ L\ w \overset{\mathsf{def}}{=} \left\{ (n,\sigma) \;\middle|\; \begin{array}{l} \mathrm{dom}(\sigma) = L \wedge \text{for all } l \in L. \\ \quad n = 0 \text{ or } (n-1, \sigma(l)) \in \mathtt{JSVal}_{\mu,\rho}\ (\mathit{roll}^{-1}_{\hat{W}}\ w) \end{array} \right\}$$

$$\mathit{RefIO}^{rgn}_{j,rec} : \mathit{EffInt} \rightarrow_{ne} \mathit{EffInt}$$

$$\mathit{RefIO}^{rgn}_{j,rec}(\rho,\mu) \overset{\mathsf{def}}{=} (\mathit{Ref}^{rgn}_{j,rec}\ (\rho,\mu), \mathit{IO}^{rgn}_{j,rec}\ (\rho,\mu))$$

Lemma 69. $\mathit{RefIO}^{rgn}_{j,rec}$ *is well-defined, i.e. it produces a result in* EffInt *for correct input arguments* $(\rho,\mu) \in \mathit{EffInt}$*. Also,* $H^{rgn}_{rec}\ (\rho,\mu)\ L$ *is effectively in* StorePred *for* $(\rho,\mu) \in \mathit{EffInt}$*.*

Proof. Uniformity and monotonicity of $\mathit{Ref}^{rgn}_{j,rec}$ follow easily from the definition. Non-expansiveness of $\mathit{Ref}^{rgn}_{j,rec}$ in w and $\mathit{IO}^{rgn}_{j,rec}$ in P and w follow from the definitions and Lemma 60, non-expansiveness of μ in P and w, Lemma 43 and Lemma 62.

Non-expansiveness and monotonicity of H^{rgn}_{rec} follow from non-expansiveness and monotonicity of $\mathtt{JSVal}_{\mu,\rho}$ and proper use of step-indices. □

Lemma 70. $\mathit{RefIO}^{rgn}_{j,rec}$ *is contractive in* (ρ,μ)*.*

Proof. Take $(\rho_1,\mu_1),(\rho_2,\mu_2) : (T \times (T \rightarrow T))$. Suppose that $(\rho_1,\mu_1) =_n (\rho_2,\mu_2)$, equivalent to $(\mu_1\ P\ w)_{[n]} = (\mu_2\ P\ w)_{[n]}$ and $(\rho_1\ w)_{[n]} = (\rho_2\ w)_{[n]}$ for all P, w (by Lemma 39). Then we need to show that $\mathit{RefIO}^{rgn}_{j,rec}\ (\rho_1,\mu_1) =_{n+1} \mathit{RefIO}^{rgn}_{j,rec}(\rho_2,\mu_2)$, which is equivalent to $(\mathit{IO}^{rgn}_{j,rec}\ (\rho_1,\mu_1)\ P\ w)_{[n+1]} = (\mathit{IO}^{rgn}_{j,rec}\ (\rho_2,\mu_2)\ P\ w)_{[n+1]}$ and $(\mathit{Ref}^{rgn}_{j,rec}\ (\rho_1,\mu_1)\ w)_{[n+1]} = (\mathit{Ref}^{rgn}_{j,rec}\ (\rho_2,\mu_2)\ w)_{[n+1]}$ for all P, w. By symmetry, it suffices to prove the inclusion of the left sets in the right sets.

First, we know from Lemma 54 that $\mathtt{JSVal}_{\mu,\rho}$ is contractive in μ and non-expansive in ρ, so that $\mathtt{JSVal}_{\mu_1,\rho_1}\ w =_n \mathtt{JSVal}_{\mu_2,\rho_2}\ w$ for all w. From this, it follows easily that $H^{rgn}_{j,rec}\ (\rho_1,\mu_1) =_{n+1} H^{rgn}_{j,rec}\ (\rho_2,\mu_2)$.

Now take $(k,e) \in (\mathit{Ref}^{rgn}_{j,rec}\ (\rho_1,\mu_1)\ w)_{[n+1]}$. That means that $k \leq n$. We need to prove that also $(k,e) \in (\mathit{Ref}^{rgn}_{j,rec}\ (\rho_2,\mu_2)\ w)$. That means that we know that $H =_{k+1} (H^{rgn}_{rec}(\rho_1,\mu_1))$ and we need to show that also $H =_{k+1} (H^{rgn}_{rec}(\rho_2,\mu_2))$. But $H^{rgn}_{rec}\ (\rho_1,\mu_1) =_{n+1} H^{rgn}_{rec}\ (\rho_2,\mu_2)$, so this is okay.

Now take $(k,e) \in (\mathit{IO}^{rgn}_{j,rec}\ (\rho_1,\mu_1)\ P\ w)_{[n]}$. That means that $k \leq n$. We need to prove that also $(k,e) \in (\mathit{IO}^{rgn}_{j,rec}\ (\rho_2,\mu_2)\ P\ w)$. If $e \in \mathit{Val}$, then we directly get that $(k,e) \in P\ w$. Now take a L and H so that $w(j) = (L, \subseteq, \subseteq, H)$ and $H =_{k+1} (H^{rgn}_{rec}(\rho_1,\mu_1))$. Since $H^{rgn}_{rec}\ (\rho_1,\mu_1) =_{n+1} H^{rgn}_{rec}\ (\rho_2,\mu_2)$, we also have that $H =_{k+1} (H^{rgn}_{rec}(\rho_2,\mu_2))$. Take $0 < i \leq k$, $\sigma_r :_k w$ and σ_f with $(\sigma_r \uplus \sigma_f, e) \rightarrow^{i}_{A} (\sigma', e')$. We get a σ'_r and $w' \sqsupseteq^{\mathrm{pub}} w$ such that $\sigma' = \sigma'_r \uplus \sigma_f$, $A \subseteq ((L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D})$, $w'(j) = (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H)$, $(k-i, e') \in \mathcal{E}[\mu_1\ P]\ w'$ and $\sigma'_r :_{k-i} w'$. We know that $k - i < k \leq n$, $\mu_1 =_n \mu_2$ and $\mathcal{E}$ non-expansive (by Lemma 43), so that $\mathcal{E}[\mu_1\ P]\ w' =_n \mathcal{E}[\mu_2\ P]\ w'$ and $(k-i, e') \in \mathcal{E}[\mu_2\ P]\ w'$. □

With $\mathit{RefIO}^{rgn}_{j,rec}$ contractive, we know it has a fixpoint satisfying the definition equation of Ref^{rgn} and IO^{rgn}. □

Lemma 71. *If $(k, e) \in \mathcal{E}[IO_j^{rgn}\ P]\ w$, then for all L and H with $w(j) = (L, \subseteq, \subseteq, H)$, $H =_{k+1} H_j^{rgn}$ and for all $i \leq k$, $\sigma_r :_k w$, σ_f with $(\sigma_r \uplus \sigma_f, e) \rightarrow_A^i (\sigma', e')$, there exists a σ'_r such that $\sigma' = \sigma'_r \uplus \sigma_f$, $A \subseteq (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$, and there exists a $w' \sqsupseteq^{\mathrm{pub}} w$ with $w'(j) = (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H)$, $\sigma'_r :_{k-i} w'$ and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P]\ w'$.*

Proof. By Lemma 36 (easily adapted to the labeled operational semantics), the evaluation $(\sigma, e) \rightarrow_A^i (\sigma', e')$ factors in one of the following ways:

- $e \rightarrow^i e'$, $A = \emptyset$, $\sigma_r \uplus \sigma_f = \sigma'$. In this case, we can conclude with $\sigma'_r = \sigma_r$, $w' = w$, since $\sigma_r :_{k-i} w$ because of Lemma 41 and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P]\ w$ because of Lemma 47.
- $e \rightarrow^{i'} e''$ and $(\sigma_r \uplus \sigma_f, e'') \rightarrow_A^{i''} (\sigma', e')$ with $e'' \in \mathit{Cmd}$, $i' + i'' = i$ and $i'' > 0$. In this case, we get that $(k - i', e'') \in IO_j^{rgn}\ P\ w$ by definition of $\mathcal{E}[\_]$. By Lemma 41, we get that $\sigma_r :_{k-i'} w$. Because $i'' > 0$ and $i'' \leq i \leq n$, we can apply the definition of IO_j^{rgn} (using $w(j) = (L, \subseteq, \subseteq, H)$ and $H =_{k+1} H_j^{rgn}$) to get a σ'_r with $\sigma' = \sigma'_r \uplus \sigma_f$, $A \subseteq (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$ and a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $w'(j) = (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H)$, $\sigma'_r :_{k-i} w'$ and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P]\ w'$.

□

Lemma 72. *For any j, IO_j^{rgn} and Ref_j^{rgn} satisfy the remaining axioms for a valid effect interpretation.*

Proof. Again, we look at all the axioms. For easy reference, we will always first re-state them with IO_j^{rgn} and Ref_j^{rgn} filled in for μ and ρ.

We use Lemma 55 and prove alternative axioms A-BINDPRIME and A-INVPURE instead of A-BIND.

- A-BINDPRIME: We need to show that if $(k, e) \in IO_j^{rgn}\ P\ w$, $e \notin \mathit{Val}$ and $(i, E\langle v\rangle) \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$ for all $i \leq k$, $w' \sqsupseteq w$, $(i, v) \in P\ w'$ then $(k, E\langle e\rangle) \in IO_j^{rgn}\ P'\ w$. We will do this by complete induction on k.
  By the definition of IO_j^{rgn}, we need to show first that $(k, E\langle e\rangle) \in P'\ w$ if $E\langle e\rangle \in \mathit{Val}$, but this is vacuously true because $E\langle e\rangle \in (\mathit{Cmd} \setminus \mathit{Val})$ by Lemma 29. Secondly, take L,H and assume that $w(j) = (L, \subseteq, \subseteq, H)$ and $H =_{k+1} H_j^{rgn}$. Take $0 < i \leq k$, $\sigma_r :_k w$, σ_f and $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow_A^i (\sigma', e')$. Then we need to produce a σ'_r with $\sigma' = \sigma'_r \uplus \sigma_f$, $A \subseteq ((L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D})$ and a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $w'(j) = (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H)$, $\sigma'_r :_{k-i} w'$ and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$.
  Take i' the largest number such that the evaluation $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow_A^i (\sigma', e')$ starts with $(\sigma_r \uplus \sigma_f, E\langle e\rangle) \rightarrow_{A'}^{i'} (\sigma'', E\langle e''\rangle)$. The sub-evaluation $(\sigma_r \uplus \sigma_f, e) \rightarrow_{A'}^{i'} (\sigma'', e'')$ together with $(k, e) \in IO_j^{rgn}\ P\ w$ and the other assumptions above gives us a σ''_r with $\sigma'' = \sigma''_r \uplus \sigma_f$, $A' \subseteq ((L \cup (\mathrm{dom}(\sigma''_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D})$ and a $w'' \sqsupseteq^{\mathrm{pub}} w$ with $w''(j) = (L \cup (\mathrm{dom}(\sigma''_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H)$, $\sigma''_r :_{k-i'} w''$ and $(k - i', e'') \in \mathcal{E}[IO_j^{rgn}\ P]\ w''$.
  If the impure evaluation stops there, then $i' = i$, $A = A'$, $\sigma' = \sigma'' = \sigma''_r \uplus \sigma_f$ and $e' = E\langle e''\rangle$. We can take $\sigma'_r = \sigma''_r$, $w' = w''$ and it only remains to show that $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$. To show this, take $i'' \leq k - i$ and $e' = E\langle e''\rangle \rightarrow^{i''} e'''$ with $e''' \in \mathit{Cmd}$. Take the largest i''' such that this pure evaluation starts with $E\langle e''\rangle \rightarrow^{i'''} E\langle e''''\rangle$. e'''' must be in Cmd because i''' is as large as possible and because of Lemmas 31 and 32. The sub-evaluation $e'' \rightarrow^{i'''} e''''$ and the fact that $(k - i, e'') \in \mathcal{E}[IO_j^{rgn}\ P]\ w'$ tells us that $(k - i - i''', e'''') \in IO_j^{rgn}\ P\ w'$. If $e'''' \in (\mathit{Cmd} \setminus \mathit{Val})$, then the pure evaluation must stop there (by Lemma 32), so $i''' = i''$ and $e''' = E\langle e''''\rangle$. We can then apply the induction hypothesis to conclude that $(k - i - i'', e''') = (k - i - i'', E\langle e''''\rangle) \in IO_j^{rgn}\ P'\ w'$. If $e'''' \in \mathit{Val}$, then $(k - i - i''', e'''') \in IO_j^{rgn}\ P\ w'$ implies that $(k - i - i''', e'''') \in P\ w'$. We can then apply the assumption about E to obtain that $(k - i - i''', E\langle e''''\rangle) \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$. With the remaining pure evaluation and the fact that $e''' \in \mathit{Cmd}$, we get that $(k - i - i'', e''') \in IO_j^{rgn}\ P'\ w'$.
  If the impure evaluation continues after evaluating e to e'', then e'' must be a value by Lemma 33. With $(k - i', e'') \in \mathcal{E}[IO_j^{rgn}\ P]\ w''$ established, this implies that $(k - i', e'') \in P\ w''$. We can now apply the assumption about E to obtain that $(k - i', E\langle e''\rangle) \in \mathcal{E}[IO_j^{rgn}\ P']\ w''$. We call $L'' = L \cup (\mathrm{dom}(\sigma''_r) \setminus \mathrm{dom}(\sigma_r))$ and we apply Lemma 71 to the remaining evaluation $(\sigma''_r \uplus \sigma_f, E\langle e''\rangle) \rightarrow_{A''}^{i-i'} (\sigma', e')$, using the fact that $\sigma''_r :_{k-i'} w''$, to obtain a σ'_r such that $\sigma' = \sigma'_r \uplus \sigma_f$, $A'' \subseteq (L'' \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma''_r))) \times \mathbb{D}$, and a $w' \sqsupseteq^{\mathrm{pub}} w''$ such that $w'(j) = (L'' \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma''_r)), \subseteq, \subseteq, H)$, $\sigma'_r :_{k-i} w'$ and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$. If we note that
$$(L'' \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma''_r))) =$$
$$(L \cup (\mathrm{dom}(\sigma''_r) \setminus \mathrm{dom}(\sigma_r)) \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma''_r))) =$$
$$(L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))$$
  and $(L \cup (\mathrm{dom}(\sigma''_r) \setminus \mathrm{dom}(\sigma_r))) \subseteq (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)))$ then we now have also in this case that $A = A' \cup A'' \subseteq (L \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$, $\sigma'_r :_{k-i} w'$ and $(k - i, e') \in \mathcal{E}[IO_j^{rgn}\ P']\ w'$ for
$$\begin{aligned} w'(j) &= (L'' \cup (\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma'')), \subseteq, \subseteq, H) \\ &= (L \cup (\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)), \subseteq, \subseteq, H). \end{aligned}$$
- A-INVPURE: $(k, e) \in IO_j^{rgn}\ P\ w$ and $e \in \mathit{Val}$ implies that $(k, e) \in P\ w$. By definition.

- A-PURE: If $v \in Val$, then $(n,v) \in P\ w$ implies $(n,v) \in IO_j^{rgn}\ P\ w$. By definition of IO_j^{rgn}, we first need to prove that if $v \in Val$, then $(n,v) \in P\ w$, which is obviously fine here. Second, take L,H and assume $w'(j) = (L, \subseteq, \subseteq, H)$ and $H =_{n+1} H_j^{rgn}$. Furthermore, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f, $(\sigma_r \uplus \sigma_f, v) \rightarrow_A^i (\sigma', e')$. By Lemma 28, the latter is not possible, so this requirement is also OK.
- A-ASSIGN: Suppose $(n, e_1) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$, $(n, e_2) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. Then $(n, e_1 = e_2) \in IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. First, it's clear that $e_1 = e_2 \notin Val$. Second, take L and H such that $w(j) = (L, \subseteq, \subseteq, H)$. Furthermore take $0 < i \leq n$, $\sigma_r :_n w$, σ_f and $(\sigma_r \uplus \sigma_f, e_1 = e_2) \rightarrow_A^i (\sigma', e')$. Because e_1 and e_2 are values, we have that the impure evaluation must start with an application of rule E-SETREF. This further implies that $e_1 = l$, $i = 1$, $A = \{(l, \mathtt{w})\}$, $e' = e_2$ and $\sigma' = \sigma[l \mapsto e_2]$. By Lemma 52 and monotonicity, we obtain that $(n, e_1) = (n, l) \in Ref_j^{rgn}\ w$, which implies that $l \in L$ and $H =_{n+1} H_j^{rgn}$. It is then clear that for $\sigma_r' = \sigma_r[l \mapsto e_2]$, we have that $\sigma' = \sigma_r' \uplus \sigma_f$ and $A \subseteq (L \cup (\mathrm{dom}(\sigma_r') \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$. Furthermore we can take $w' = w$ and it remains to be proven that $\sigma_r' :_{n-1} w$ and $(n-1, e_2) \in \mathcal{E}[IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$. The latter follows directly from the uniformity of $\mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}$, the already proven Axiom A-PURE and Lemma 48. The former follows because $\sigma_r :_n w$, $\sigma_r' = \sigma_r[l \mapsto e_2]$ and $(n-1, e_2) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$ (by uniformity) and from the fact that $H =_{n+1} H_j^{rgn}$.
- A-REF: Suppose $(n, e) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. $(n, \mathtt{ref}\ e) \in IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. First, it's clear that $\mathtt{ref}\ e \notin Val$. Second, take L and H such that $w(j) = (L, \subseteq, \subseteq, H)$ and $H =_{n+1} H_j^{rgn}$. Furthermore, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f such that $(\sigma_r \uplus \sigma_f, \mathtt{ref}\ e) \rightarrow_A^i (\sigma', e')$. Because e is a value, we have that the impure evaluation must start with an application of rule E-REF. This further implies that $e' = l$ for some $l \notin \mathrm{dom}(\sigma)$, $i = 1$, $A = \{(l, \mathtt{r}), (l, \mathtt{w})\}$, $\sigma' = \sigma[l \mapsto e]$. We have that $l \in \mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)$, so we can take $\sigma_r' = \sigma_r[l \mapsto e]$, and it is clear that $\sigma' = \sigma_r' \uplus \sigma_f$ and $A \subseteq (L \cup (\mathrm{dom}(\sigma_r') \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$. Furthermore we take $L' = L \cup \{l\}$ and $w' = w[j \mapsto (L', \subseteq, \subseteq, H)]$ and it remains to show that $\sigma_r' :_{n-1} w'$ and $(n-1, l) \in \mathcal{E}[IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w'$. The latter follows because $H =_{n+1} H_j^{rgn}$, so that definitely $(n-1, l) \in Ref_j^{rgn}\ w'$, and because $Ref_j^{rgn}\ w' \subseteq \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w'$ by the definition of $\mathtt{JSVal}_{\mu,\rho}$, because of already proven Axiom A-PURE and Lemma 49. The former follows by definition of $\sigma_r' :_{n-1} w'$, by the fact that $\sigma_r' = \sigma_r[l \mapsto e]$, $\sigma_r :_n w$ and $(n, e) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$, by monotonicity and uniformity and by the fact that $H =_{n+1} H_j^{rgn}$.
- A-DEREF: Suppose $(n, e) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. Then $(n, \mathtt{deref}\ e) \in IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. First, it's clear that $\mathtt{deref}\ e \notin Val$. Second, take L and H such that $w(j) = (L, \subseteq, \subseteq, H)$ and $H =_{n+1} H_j^{rgn}$. Furthermore, take $0 < i \leq n$, $\sigma_r :_n w$, σ_f and $(\sigma_r \uplus \sigma_f, \mathtt{deref}\ e) \rightarrow_A^i (\sigma', e')$. Because e is a value, we have that the impure evaluation must start with an application of rule E-DEREF. This further implies that $e = l$, $i = 1$, $A = \{(l, \mathtt{r})\}$, $e' = h(l)$ and $\sigma' = \sigma_r \uplus \sigma_f$. By Lemma 52 and monotonicity, we obtain that $(n, e) = (n, l) \in Ref_j^{rgn}\ w$, which implies that $l \in L \subseteq \mathrm{dom}(\sigma_r)$. So we can take $\sigma_r' = \sigma_r$ and it is then clear that $\sigma' = \sigma_r' \uplus \sigma_f$ and $A \subseteq (L \cup (\mathrm{dom}(\sigma_r') \setminus \mathrm{dom}(\sigma_r))) \times \mathbb{D}$. We can take $w' = w$ and it remains to be proven that $\sigma_r' :_{n-1} w$ and $(n-1, \sigma(l)) \in \mathcal{E}[IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$. The former follows from the fact that $\sigma_r :_n w$, and Lemma 41. The latter follows from $\sigma_r :_n w$ which implies that $(n-1, \sigma_r(l)) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$, Axiom A-PURE and Lemma 48.

□

The next lemma connects Maffeis et al.'s tCap function to our notion of well-scopedness.

Lemma 73. $\Sigma \supseteq \mathtt{tCap}(e)$ *for a closed expression* e *iff* $\emptyset; \Sigma \vdash e$.

Proof. Induction on e. □

Next, we need to prove some things about worlds in $[\![\Sigma]\!]_{IO_j^{rgn}, Ref_j^{rgn}}$.

Lemma 74.
- $(k, w) \in [\![\Sigma]\!]_{IO_j^{rgn}, Ref_j^{rgn}}$ *iff* $w(j) = (L, \subseteq, \subseteq, H)$, $H =_{k+1} H_j^{rgn}$ *and* $\Sigma \subseteq L$.
- *If* $\mathtt{tCap}(\sigma(l)) \subseteq \Sigma$ *for all* $l \in \Sigma$, *then there are* σ_r *and* σ', w *and* j *such that* $\sigma = \sigma_r \uplus \sigma'$, $\sigma_r :_k w$ *for all* k *and* $w(j) = (\Sigma, \subseteq, \subseteq, H_j^{rgn})$.

Proof.
- $(k, w) \in [\![\Sigma]\!]_{IO_j^{rgn}, Ref_j^{rgn}}$ iff for all $l \in \Sigma$, we have that $(k, l) \in Ref_j^{rgn}\ w$. The latter is true iff $w(j) = (L, \subseteq, \subseteq, H)$, $H =_{k+1} H_j^{rgn}$ and $l \in L$.
- Take $j = 1$, $w = (1, \iota_j^{rgn})$. We take $\sigma_r = \sigma|_\Sigma$ and σ' such that $\sigma = \sigma_r \uplus \sigma'$. Take k arbitrary. To prove that $\sigma_r :_k w$, we need to show that $(k, \sigma_r) \in H_j^{rgn}\ \Sigma\ (roll\ w)$, i.e. that for an arbitrary $l \in \Sigma$ $k = 0$ or $(k-1, \sigma(l)) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. We know that $\mathrm{tCap}(\sigma(l)) \subseteq \Sigma$, so Lemma 73 tells us that $\emptyset; \Sigma \vdash \sigma(l)$. The Fundamental Theorem (Theorem 3) with Lemma 72 then tells us that this implies $(k-1, \sigma(l)) \in \mathcal{E}[IO_j^{rgn}\ \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$ because $(k-1, w) \in [\![\Sigma]\!]_{IO_j^{rgn}, Ref_j^{rgn}}$ (see above). But we know that $\sigma(l) \in Val$ for all l, so the definitions of $\mathcal{E}[t]$ and IO_j^{rgn} give us that $(k, \sigma(l)) \in \mathtt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$.

□

Lemma 4. *If* $\mathrm{auth}(\sigma, e) = L \times \mathbb{D}$*, then* $\texttt{memBound}(\sigma, e, L)$.

Proof. By definition of auth, we have that for all $l \in L$, $\texttt{tCap}(\sigma(l)) \subseteq L$. Lemma 74 then gives us σ_r, σ', w and j such that $\sigma = \sigma_r \uplus \sigma'$, $\sigma_r :_n w$ for all n and $w(j) = (L, \subseteq, \subseteq, H_j^{rgn})$. It remains to prove that for an arbitrary n, $(n, e) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$. By definition of auth, we have that $\texttt{tCap}(e) \times \mathbb{D} \subseteq L \times \mathbb{D} = \mathrm{auth}(\sigma, e)$. Now combine Lemma 73, the Fundamental Theorem of logical relations (Theorem 3), Lemma 74 and Lemma 72. $\square$

Lemma 5. *If* $\texttt{memBound}(\sigma_1, e_1, L)$ *and* $(\sigma_1, e_1) \rightarrow_A^i (\sigma_2, e_2)$*, then for* $L' \overset{\text{def}}{=} L \cup (\mathrm{dom}(\sigma_2) \setminus \mathrm{dom}(\sigma_1))$*, we have that*

- $A \subseteq L' \times \mathbb{D}$ *and* $\sigma_2|_{\mathrm{dom}(\sigma_1)\setminus L} = \sigma_1|_{\mathrm{dom}(\sigma_1)\setminus L}$.
- $\texttt{memBound}(\sigma_2, e_2, L')$ *with*

Proof. We prove the two statements together. Take an arbitrary n. We apply the definition of $\texttt{memBound}$ for $n+i$, to obtain for $w = (1, \iota_{j,L}^{rgn})$ and $j = 1$ that $(n+i, e_1) \in \mathcal{E}[IO_j^{mem}\ \texttt{JSVal}_{IO_j^{mem}, Ref_j^{mem}}]\ w$ and a σ_f and $\sigma_{1,r}$ such that $\sigma_1 = \sigma_{1,r} \uplus \sigma_f$ and $\sigma_{1,r} :_{n+i} w$ and . Lemma 71 for $w' = w$ gives us a $\sigma_{2,r}$ such that $\sigma_2 = \sigma_{2,r} \uplus \sigma_f$, $A \subseteq L' \times \mathbb{D}$ and gives us a $w' \sqsupseteq^{\mathrm{pub}} w$ such that $w'(j) = (L', \subseteq, \subseteq, H_j^{rgn})$, $\sigma_{2,r} :_k w'$ and $(n, e_2) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w'$. Note that it's clear that $\mathrm{dom}(\sigma_{2,r}) \setminus \mathrm{dom}(\sigma_{1,r}) = \mathrm{dom}(\sigma_2) \setminus \mathrm{dom}(\sigma_1)$ from the equations for σ_1 and σ_2 above. Also, the facts that $\sigma_{2,r} :_k w'$, $w'(j) = (L', \subseteq, \subseteq, H_j^{rgn})$, $\sigma_{1,r} :_{n+i} w$, $w = (1, \iota_{j,L}^{rgn})$ and $w' \sqsupseteq^{\mathrm{pub}} w$ imply that $L = \mathrm{dom}(\sigma_{1,r})$, $L' = \mathrm{dom}(\sigma_{2,r})$ and $w' = (1, \iota_{j,L'}^{rgn})$. Because we started with an arbitrary n, we can conclude that $\texttt{memBound}(\sigma_2, e_2, L')$. $\square$

Lemma 6. *If* $(\sigma, e) \rightarrow_A^* (\sigma', v)$*,* $A \not\triangleright L \times \mathbb{D}$ *and* $\texttt{memBound}(\sigma, e', L)$*, then still* $\texttt{memBound}(\sigma', e', L)$.

Proof. Take an arbitrary n. $\texttt{memBound}(\sigma, e', L)$ gives us for $j = 1$ and $w = (1, \iota_{j,L}^{rgn})$ that $(n, e') \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$ and σ_f, σ_r such that $\sigma_r :_n\ w$.

We prove that $\texttt{memBound}(\sigma', e', L)$. We directly have that $(n, e') \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$. It remains to produce σ'_f and σ'_r such that $\sigma' = \sigma'_r \uplus \sigma'_f$ and $\sigma'_r :_n w$.

It follows from $A \not\triangleright L \times \mathbb{D}$, that $\{l \mid (l, \mathrm{w}) \in \texttt{acc}(\tau(\sigma, e))\} \cap L = \emptyset$ and as a consequence $\sigma|_L = \sigma'|_L$ and $(\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)) \cap L = \emptyset$. From $\sigma_r :_n w$, we get that $\mathrm{dom}(\sigma_r) = L$, so that we can take $\sigma'_r = \sigma_r$ and some σ'_f such that $\sigma' = \sigma'_r \uplus \sigma'_f$. We then already have that $\sigma'_r :_n w$. $\square$

Lemma 7. *Suppose* $\texttt{memBound}(\sigma, e_1, L_1)$ *and* $\texttt{memBound}(\sigma, e_2, L_2)$*. Suppose that* $(\sigma, e_1) \rightarrow_A^* (\sigma', v)$ *and* $A \triangleright L_2 \times \mathbb{D}$*. Define* $L' \overset{\text{def}}{=} L_1 \cup L_2 \cup (\mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma))$*. Then* $\texttt{memBound}(\sigma', e_2, L')$.

Proof. From $\texttt{memBound}(\sigma, e_1, L_1)$, we get for $j = 1$, $w_1 = (1, \iota_{j,L_1}^{rgn})$ and i the number of steps in the evaluation $(\sigma, e_1) \rightarrow^* (\sigma', v)$, that $(n+i, e_1) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w_1$ and some $\sigma_{1,f}$ and $\sigma_{1,r}$ such that $\sigma = \sigma_{1,r} \uplus \sigma_{1,f}$ and $\sigma_{1,r} :_{n+i} w_1$.

Lemma 71 then tells us for $w' = w_1$ and $L = L_1$ that there exist $\sigma'_{1,r}$ and $w'_1 \sqsupseteq^{\mathrm{pub}} w_1$ such that $\sigma' = \sigma'_{1,r} \uplus \sigma_{1,f}$, $w'_1(j) = (L_1 \cup (\mathrm{dom}(\sigma'_r) \setminus \mathrm{dom}(\sigma_r)), \subseteq, \subseteq, H_j^{rgn})$, $\sigma'_{1,r} :_n w'_1$ and $(n, v) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w'_1$. Write $L'_1 = L_1 \cup (\mathrm{dom}(\sigma'_{1,r}) \setminus \mathrm{dom}(\sigma_{1,r}))$. Because $\sigma_{1,r} :_{n+i} w_1$, we have that $\mathrm{dom}(\sigma_{1,r}) = L_1$. From $w'_1(j) = (L'_1, \subseteq, \subseteq, H_j^{rgn})$ and $\sigma'_{1,r} :_n w'_1$, we can then deduce that $w'_1 = (1, \iota_{j,L'_1}^{rgn})$.

From $\texttt{memBound}(\sigma, e_2, L_2)$, we get for $j = 1$, $w_2 = (1, \iota_{j,L_2}^{rgn})$ that $(n, e_2) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w_2$ and some $\sigma_{2,f}$ and $\sigma_{2,r}$ such that $\sigma = \sigma_{2,r} \uplus \sigma_{2,f}$ and $\sigma_{2,r} :_n w_2$. Note that it follows that $\mathrm{dom}(\sigma_{2,r}) = L_2$.

Now take $w = (1, \iota_{j,L'}^{rgn})$. It's clear that $w \sqsupseteq w_2$, so we get from uniformity that $(n, e_2) \in \mathcal{E}[IO_j^{rgn}\ \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}]\ w$. Now take $L'_2 = L_2 \setminus L'_1$, $\sigma_r = \sigma'_{1,r} \cup \sigma_{1,f}|_{L'_2}$ and $\sigma_f = \sigma|_{\mathrm{dom}(\sigma)\setminus L}$. It follows from $\sigma' = \sigma'_{1,r} \uplus \sigma_{1,f}$, $\sigma = \sigma_{1,r} \uplus \sigma_{1,f}$, the definition of L and the domains of $\sigma_{1,r}$ and $\sigma_{1,f}$ that $\sigma' = \sigma_r \uplus \sigma_f$. It remains to prove that $\sigma_r :_n w$.

This means that we have to show that $\sigma_r \in H_j^{rgn}\ L'\ (\mathit{roll}\ w)$, i.e. that for all $l \in L'$, $(n, \sigma_r(l)) \in \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$. So take $l \in L'$, and distinguish the following two cases:

- $l \in L'_1$: then we know that $\sigma_r(l) = \sigma'_{1,r}(l)$, $\sigma'_{1,r} :_n w'_1$ and $w \sqsupseteq w'_1$, so that $(n, \sigma_r(l)) \in \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$ follows by monotonicity.
- $l \in L'_2$: here we know that $\sigma_r(l) = \sigma_{1,f}(l) = \sigma(l) = \sigma_{2,r}(l)$, $\sigma_{2,r} :_n w_2$ and $w \sqsupseteq w_2$, so that $(n, \sigma_r(l)) \in \texttt{JSVal}_{IO_j^{rgn}, Ref_j^{rgn}}\ w$ follows by monotonicity.

$\square$