Access Permission Contracts for Scripting Languages

Phillip Heidegger Annette Bieniusa Peter Thiemann

University of Freiburg, Germany

{heidegger,bieniusa,thiemann}@informatik.uni-freiburg.de

Abstract

The ideal software contract fully specifies the behavior of an operation. Often, in particular in the context of scripting languages, a full specification may be cumbersome to state and may not even be desired. In such cases, a partial specification, which describes selected aspects of the behavior, may be used to raise the confidence in an implementation of the operation to a reasonable level.

We propose a novel kind of contract for object-based languages that specifies the side effects of an operation with *access permissions*. An access permission contract uses sets of access paths to express read and write permissions for the properties of the objects accessible from the operation.

We specify a monitoring semantics for access permission contracts and implement this semantics in a contract system for JavaScript. We prove soundness and stability of violation under increasing aliasing for our semantics.

Applications of access permission contracts include security, enforcing modularity, test-driven development, program understanding, and regression testing. With respect to testing and understanding, we find that adding access permissions to contracts increases the effectiveness of error detection through contract monitoring by 6-13%.

1. Introduction

Design by contract is a methodology for software development based on specifications (contracts) of operations [35, 36]. The correctness of an implementation with respect to a contract may be statically guaranteed by program verification or it may be dynamically checked with contract monitoring. As the latter variant permits more expressive specifications and puts less demands on the theorem proving skills of the programmer, it is widely used in practice as evidenced by implementations of contract checking in various forms and for many languages [1, 14, 15, 16, 17, 24, 27, 29, 44].

Originally, contracts were meant to provide full specifications. However, contracts for partial specifications, which only fix certain aspects of an operation, also have their uses. For example, in a dynamically-typed language, like Scheme or JavaScript, a contract could have the form of an expressive type signature and impose restrictions similar to a type system [2, 24, 43]. Contract monitoring for such type contracts detects type errors at operation boundaries.

A type contract has one important drawback. It only imposes restrictions on the values passed to an operation and returned from it. In an imperative language like Scheme or JavaScript, many operations have an effect on the heap, which is not captured by a type contract. For those operations, a contract that also specifies the effect would be more appropriate.

In the past, this drawback has driven the evolution of type systems towards effect systems that enable the specification and inference of side effects (e.g., [19, 42]). In analogy, we propose to extend type contracts with a language-dependent notion of effects and to check them with an application-dependent notion of monitoring. In this paper, we develop a notion of effects suitable for scripting languages and for JavaScript in particular. The application scenarios that we have in mind are security, enforcing modularity, test-driven development, program understanding, and regression testing.

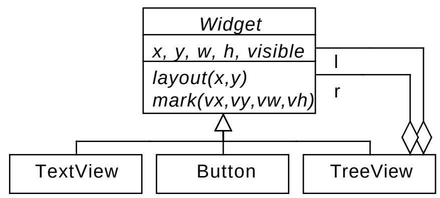

Figure 1. Example widget hierarchy.

1.1 Effects for Scripting Languages

Like other scripting languages, JavaScript implements an object as a property-to-value map. Thus, every read or write access to an object graph can be described by a *base object*, a *path* (a sequence of property names), and a *classifier* indicating the operation on the last step of the path (read or write).

This observation has lead us to define effects by *access permissions* that specify a set of paths that an operation may read or write relative to some base object in scope. Most of the time, the base object is this or an argument of the operation, but a global variable may also be used.

As an example, consider a JavaScript implementation of the widget class hierarchy described by the class diagram in Fig. 1. The layout operation computes the screen position of each widget. It accepts a pair of absolute starting coordinates and returns the width and height of the rendered widget. As a side effect, it stores the bounding box of each subwidget in its representation. A programmer working on the code of this operation might like to ascertain that the layout computation only ever changes the bounding box properties of a widget by attaching the following contract to the layout operation:

(int, int) → {w: int, h: int} **with** [this.∗./x|y|w|h/]

This contract specifies an operation that accepts two integers, returns an object with two integer properties named w and h, and modifies at most the x, y, w, and h properties of objects reachable through this. Furthermore, the contract allows the operation to read *all* properties reachable through this: a write path has to match the entire access permission, whereas a read path is accepted if it matches a prefix of the permission. More precisely, this specifies the base object, "." separates the path components, ∗ matches any sequence of property names, and /x|y|w|h/ is a regular expression that matches the names of the properties with write permission.

In the implementation, a set of properties can be specified with a JavaScript regular expression and any such set can be iterated using the $*$ operator. For example, granting permission to access any property of window except location may be expressed by ... **with** [window./^((?!location$).)/].

1.2 Application Scenarios

Regarding the application scenarios, contract monitoring for the layout contract is useful during the initial **test-driven development** of the code because any violation of the access permission would trigger an exception as part of a test run. It is also useful for **program understanding**. A programmer who would like to confirm that the layout operation works in the outlined way would impose the contract and watch for failing test cases. The access permissions also indicate what operations are independent of one another. For example, an operation that marks those widgets which are visible in a given viewport might have a contract like this:

```
({vx: int, vy: int, vw: int, vh: int}) → void
    with [$1.?.@, this./l|r/*./x|y|w|h/.@, this./l|r/*.visible]
```

It expresses that any property[1] of the first parameter[2] may be read, but not written[3]; only the bounding box of a widget may be read; and only its visible property may be written. Only l and r properties may be traversed recursively. In **regression testing**, changes to the code that violate the contract would be detected early in a run of a test suite, assuming sufficient coverage.

For a **security** scenario, consider that Web browsers maintain a number of "magic properties" where an assignment causes a significant change of the browser's state (for example, window.location) or where a read operation may unveil sensitive information of the user (for example, document.cookie). Wrapping a monitored contract around a suspicious piece of code can easily reveal and prevent this kind of problem.

Last, but not least, **modularity**: JavaScript programs often rely on a number of libraries and freely include third-party code (mashups) that may change arbitrarily between different program runs. Programmers do not want this code to corrupt their global variables or to inflict arbitrary changes on their object structures. Wrapping a monitored contract around the third-party code confines these effects and guarantees the integrity of the program's state.

1.3 Monitoring of Effects

There are two approaches to defining a semantics of monitoring for an access permission. The *location-based semantics* keeps track of a set of object locations and their permissions. It traverses the object graph starting from the base object according to the access paths and registers a read or write permission (according to the path's classification) for each object property along the path. In contrast, a *path-based semantics* keeps track of access paths at run time and computes the permission at a read/write access based on the path through which the property is reached.

Both semantics have strengths and weaknesses. In the absence of aliasing, the location-based semantics is equivalent to the path-based semantics. In the presence of aliasing, the semantics differ and thorough understanding is required to predict the behavior. The following example highlights the differences. It is further elaborated in Sec. 2.1.

Suppose an object is reachable from the base object via two different paths, where one path grants write permission for property p, but a second path grants read permission, only. In the path-based semantics, the path used to access the object determines the permission, but which permission should be granted by the location-based semantics? If p gets write permission, then an execution that arrives at the object via the read path can write. If p only gets read permission, then an execution that arrives at the object via the write path cannot write. As accesses that are not mentioned in the permission are forbidden by default, the only possibility is to assign each property the least restrictive permission of all reaching paths. In this case, the location-based semantics would assign write permission, but the resulting behavior may not be intuitive. A similar dilemma arises if the second path grants access to the object containing p, but not to p itself.

Both location- and path-based semantics are non-trivial to implement efficiently. With the location-based semantics, the installation of a contract requires that the locations of all objects reachable through the contract's access paths must be marked with access rights. This marking cannot be delayed because aliasing may provide a shortcut into a data structure on which the contract grants read or write permission. In the worst case, the time needed to install a contract is unbounded because it may require a traversal of the entire object graph. A read or write operation can be implemented in almost constant time.

In contrast, the path-based semantics requires time linear in the number of installed contracts for each read and write operation, whereas the installation of a contract takes constant time. Read operations are a bit tricky because they have to juggle access paths in the right way (see Sec. 3.2).

We have chosen the path-based semantics because its semantics enjoys a number of desirable properties (see Sec.2) and its implementation cost in terms of run time is reasonable. Furthermore, we found that it is sufficiently flexible to support all intended application scenarios.

Contributions

1. Design of a contract framework with access permissions.
2. Specification of a path-based formal semantics of access permissions and their dynamic enforcement (monitoring).
3. Formal proof that the semantics guarantees stability of access violations under addition of aliasing, subject to mild conditions.
4. Prototype implementation of access permissions with monitoring in a contract and testing framework for JavaScript.The implementation is based on program transformation.
5. Assessment of the effectiveness of access permission contracts by observing the impact of random code modifications on hand-annotated case studies.
6. Practical evaluation of the approach on different code bases.

Outline

In Sec.2, we explain the design principles underlying our contract framework with examples and explore some of the alternatives and consequences. Section 3 presents a formal framework for reasoning about access permissions. It presents an operational semantics of contract monitoring and formally defines and proves two properties that underline the design principles. Section 4 explains the basic approach taken by the implementation. The evaluation in Sec. 5 explores the effectiveness of access permission for detecting programming errors using mutation testing. Section 6 outlines the extensions necessary to reliably support the security application. A discussion of related work (Sec. 7) leads to the conclusions of the paper (Sec. 8).

This is the technical report that extends the submitted paper by an appendix with further examples, a definition of the location-based semantics, proofs of all theorems, and additional information regarding the case studies.

[1] The symbol ? stands for the regular expression /.*/.

[2] Parameters may be referenced by name or by position using $1, $2, ...

[3] A path ending in @ indicates a read-only path.

2. Design Principles

The design of our monitoring semantics obeys four principles.

Path-Based Semantics An access permission grants the right to read or modify a property of an object depending on the path through which the object was reached.

Dynamic Extent An access permission for a function is in force for the duration of a function activation.

Pre-State Snapshot An access permission applies to objects and paths in the heap at the time the contract is installed.

Last Writer Wins The last write operation to a property determines the access rights for future readers of the property.

These principles explain the behavior of the semantics in corner cases involving aliasing. In the vast majority of uses, the programmer can work from the intuition provided in Sec. 1.

This section explains the principles, gives a critical overview of the alternatives, and thus provides a rationale for the choices in our framework. These principles are by and large motivated by work on static effect systems [19, 22].

2.1 Path-Based Semantics

> An access permission grants the right to read or modify a property of an object depending on the path through which the object was reached.

This principle has two consequences.

Reference Attachment Permissions are attached to individual references, not to heap locations. That is, if two variables or properties hold a reference to the same object, then accesses through each variable may have different access rights.

Stability of Violation An access violation is preserved under increased aliasing. See Sec. 3.5 for a formal statement.

Section 1.3 already argues in favor of the path-based semantics. Here, we give a concrete example where the location-based semantics behaves unexpectedly.

```
1 /*c ({}, {}) → any with [x.b,y.a] */
2 function h(x, y) {
3   y.a = 1;
4   y.b = 2; // violation?
5 }
6 function h1() {
7   var o = { a: −1, b: −2 };
8   h(o, o);
9 }
```

A static analysis would find that any invocation of function h should result in an access violation because the contract disallows accessing y.b in line 4.

The *path-based semantics* is consistent with such an analysis: Any invocation of h —in particular the call from h1— triggers a contract violation regardless of the aliasing among the arguments. We call this behavior **stability of violation**.

With the *location-based semantics*, h1 would *not* trigger a violation. As x and y are aliased, their underlying location would obtain permission to write properties a and b and the two assignments would go through without violation. Furthermore, the location-based semantics breaks stability of violation. Calling h *without aliasing* as in h2 detects a violation.

```
10 function h2() {
11   h({ a: −1, b: −2 }, { a: −1, b: −2 });
12 }
```

In a security setting, the location-based semantics appears better suited because a permission like window./^((?!location$).)/ seems to rule out any access to the location property of the window object, even if this object is reached via some alias. However, the following example demonstrates that this appearance is deceptive:

```
13 /*c ({}, any) → any with [x.?,window./^((?!location$).)/] */
14 function k(x, y) {
15   x.location = y; // violation?
16 }
17 function k1() {
18   k(window, "http://www.evil.com/");
19 }
```

As x and window are aliases of one another, the permission x.? grants write permission for all properties of window and the permission window./^((?!location$).)/ grants write permission for all properties of window, except location. Hence, the location-based semantics permits writing to window.location in the body of k.

Suffice it to say that the path-based semantics does not trigger a violation, either. However, there are extensions to both semantics that grant reliable write protection for window.location. For the path-based semantics, it requires applying a contract to the window object at the beginning of the program run before any alias is taken. Section 6 discusses the required extensions in detail.

We adopt the path-based semantics because it deals satisfactorily with the intended applications, it guarantees stability of violation, its behavior matches the intuition of the average programmer, and it is reasonably efficient. Although single read and write accesses are more expensive than for the location-based semantics, we believe that the potentially unbounded time for installing a location-based permission is not amortized by the subsequent exploration of the object graph: We expect that only a small fragment of the objects affected by a contract is actually explored.

2.2 Dynamic Extent

> An access permission for a function is in force for the duration of a function activation.

An instance of the permission gets installed at each invocation of the function and this same instance is withdrawn at the matching function return.

As a consequence, permissions get refined in a chain of function calls. Because an access permission expresses the promise that the contracted function does not exceed its permissions, each additional function call can only restrict the accessible properties further. For example, consider these functions:

```
20 /*c ({}) → any with [x.a] */
21 function d1(x) {
22   return x.a; // violation if called from d2
23 }
24 /*c ({}) → any with [] */
25 function d2(x) {
26   return d1(x);
27 }
```

The contract of function d2 disallows *any* access to its argument. Invoking d2 with any object triggers a violation of d2's contract when trying to execute line 22, although this line is in d1, which has a more permissive contract.

As another consequence, a closure returned from a function is not restricted by the access permission of the function. For example, consider the permission of f in this code fragment:

```
28 /*c ({}) → (() → any) with [x.b] */
29 function f(x) {
30   return function() { return x.a + x.b; };
31 }
32 function f1() {
```

```
33   var r = f({ a: "secret", b: "public" });
34   r();
35 }
```

Running the function f1 does not violate the contract of f because the access to x.a happens outside the dynamic extent of the call f(o).

The dynamic extent principle is inspired by the distinction between direct and latent effects in static effect systems. Evaluation of the function expression in line 30 does not cause an access to x.a or x.b. For this reason, a static effect system categorizes this effect as a *latent effect* and places it on top of the function arrow in its type. When the function is applied, the effect is exercised. However, at that point, there is no contract in force that would restrict the effect.

On the other hand, the following variant of the program fragment leads to a violation.

```
36 function g(x) {
37   return function() { return x.a + x.b; };
38 }
39 /*c ({}) → any with [x.b] */
40 function g1(x) {
41   var r = g(x);
42   r(); // violation
43 }
```

As the invocation of r() happens in the extent of the invocation of g1, its permission **with** [x.b] is in force and the violation by accessing x.a is detected. In contrast, the function g by itself does not impose any restriction on accesses to x, so that a direct call to g does not lead to a violation.

Another way to protect a property would be to add latent access permissions to the language:

```
44 /*c ({}) → (() → any with [x.b]) with [x.b] */
45 function j(x) {
46   return function() { return x.a + x.b; };
47 }
48 function j1() {
49   var r = j({ a: "secret", b: "public" });
50   r(); // violation
51 }
```

Such latent permissions are supported by our implementation, but they are not available in the syntax.

An alternative design would consider access permissions as contagious and have closures capture the contracts in force at their definition site. This design would flag both examples as violations. We dismissed this alternative because its behavior does not coincide with the static information provided by an effect system and because of the implementation cost of storing contracts in closures.

2.3 Pre-State Snapshot

> An access permission applies to objects and paths in the heap at the time the contract is installed.

One immediate implication of this principle is that the program can access and modify newly allocated objects without restriction. As these objects are not present in the heap snapshot at installation time (the pre-state), the contract does not apply to them. This behavior is analogous to the treatment of the assignable clause and newly allocated objects in JML [32, 38].

We see two alternatives to this design but consider neither viable: choose a different reference heap or choose a different interpretation of access paths.

The only other reference heap for a contract would be the post-state of a function, that is, the heap at the time the contract is withdrawn. However, the final heap is not a sensible choice because the programmer expects the paths in a contract to refer to the situation at the time a function is invoked. The final heap may exhibit very different paths.

Another interpretation of access paths might consider the permissions as symbolic paths that may be traversed regardless of the changes in the underlying heap. This interpretation also violates the programmer's intuition as the following example shows.

```
52 /*c ({}, {}) → any with [x.a, y.a, y.a.b]
53 function b(x, y) {
54   y.a = x.a;
55   y.a.b = 42; // allowed?
56 }
```

Reading just the contract, a programmer expects that x.a.b does not change. However, the symbolic interpretation of paths would not flag the assignment in line 55, which changes x.a.b, counter to the expectation of the programmer. In contrast, our proposed interpretation reports a violation.

2.4 Last Writer Wins

> The last write operation to a property determines the access rights for future readers of the property.

This principle could be considered as a consequence of the pre-state principle. It implies that aliasing created after the installation of an access contract creates new permitted access paths. Aliasing created before is not considered.

Here is an example.

```
57 /*c ({}) → any with [x.a,x.b.a] */
58 function l(x) {
59   x.a = x.b;
60   x.a.a = 42;
61 }
62 function l1() {
63   var x = { a: {}, b: {}};
64   l(x);
65 }
```

In this code fragment, line 59 is clearly permitted as x.a may be assigned to and x.b may be read. The following read access to x.a in line 60 returns the reference to the object that was accessible through x.b when the permission was installed. As this object was first reached via x.b, the access permission for x.b counts so that the assignment to x.a.a is sanctioned by the path x.b.a. Thus, function l1() runs without violation!

If we modify the example to create the alias **before** installing the permission, then things look different.

```
66 /*c ({}) → any with [x.a,x.b.a] */
67 function m(x) {
68   var y = x.a;
69   y.a = 42; // violation
70 }
71 function m1() {
72   var x = { a: {}, b: {}};
73   x.a = x.b;
74   m(x);
75 }
```

In this case, running m1() yields a violation. While the first read access to x.a in line 69 is sanctioned by x.a, the write access to property a of this object is not. Indeed, this behavior is consistent with invoking m on an object without any aliasing, which reports a violation under any semantics.[4]

3. Formalization

A formal semantics of monitoring for access permissions is needed as basis of an implementation that observes the design principles.

[4] The location-based semantics runs both examples, l1 and m1, without triggering a violation.

variable	x	$\in$	Var
property name	p	$\in$	Prop
access path	π	$\in$	$\mathit{Path} = \mathit{Prop}^*$
path language	L	$\in$	$\mathit{PLang} = \wp(\mathit{Path})$
expression	e	$::=$	$x \mid \lambda x.e \mid e(e)$
		$\mid$	$\mathtt{new} \mid e.p \mid e.p := e$
		$\mid$	$\mathtt{permit}\, x : L_r, L_w\, \mathtt{in}\, e$

Figure 2. Syntax.

For that reason, we define the calculus λ_{obj}^{AP} as a call-by-value lambda calculus extended with objects and access permissions. For this calculus, we specify the semantics, including monitoring, and prove that it adheres to the principles stated in Sec. 2, in particular, pre-state snapshot and stability of violation.

Let's fix some notation before we start. Let A and B be sets. We write $\wp(A)$ for the power set of A, $A + B$ for the disjoint union of A and B, and $A \times B$ for their Cartesian product. $A \rightharpoonup B$ denotes the set of finite (partial) functions from A to B with $\emptyset$ standing for the empty mapping and if $f \in A \rightharpoonup B$, then $f\downarrow_{A'}$ denotes the restriction of f to domain $A' \subseteq A$, $\mathit{dom}(f) \subseteq A$ denotes the domain of f and $\mathit{ran}(f) \subseteq B$ its range. The updated function $f' = f[a \mapsto b]$ is defined by $f'(a) = b$ and $f'(a') = f(a')$, for all $a' \neq a$. We also write $[a \mapsto b] = \emptyset[a \mapsto b]$ for the singleton map with domain $\{a\}$. If we write $f(a)$ as part of a premise, this use implies the additional premise $a \in \mathit{dom}(f)$.

3.1 Syntax

Figure 2 specifies the syntax of λ_{obj}^{AP}. The calculus extends a call-by-value lambda calculus with object construction (`new` creates a fresh object devoid of properties), reading of an object's property, and writing/defining an object's property. The syntax is close to that of existing JavaScript core languages [23, 25].

The novel construct of the calculus is the access permission expression $\mathtt{permit}\, x : L_r, L_w\, \mathtt{in}\, e$ that restricts accesses through variable x during evaluation of e governed by the two languages L_r and L_w. Both languages specify a set of access paths (sequences of properties) starting from the object bound to x (which must be in scope). Read accesses to descendants of x are limited to paths in L_r whereas write accesses are limited to paths in L_w. Evaluation of e stops if it tries to perform any access that is not permitted.

The read language L_r should be prefix closed, because it does not make sense to permit reading of $x.a.b$ without permitting to read $x.a$, too. Similarly, writing to $x.a.b$ is not possible without reading $x.a$, first. So, each path in the write language L_w should extend a path in the read language by one property, that is, $L_w \subseteq \{\pi.p \mid \pi \in L_r, p \in \mathit{Prop}\}$.

Our implementation restricts L_r and L_w to regular languages so that membership is decidable. Furthermore, contracts with access permissions can only be attached to functions and a contract can state multiple `permit`s in one go.

3.2 Semantics

Figure 3 defines the semantic domains and the inference rules for a big-step evaluation judgment of the form

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; v$$

This judgment declares that given a variable environment ρ and indexed collections $\mathcal{R}$ and $\mathcal{W}$ of read and write permissions, the expression e transforms the initial heap H to the final heap H' and returns value v. Furthermore, it threads a time stamp $u, u' \in \mathit{Stamp}$ that is incremented at each property write operation and at each `permit` expression. The permissions $\mathcal{R}$ and $\mathcal{W}$ are indexed by the time stamps of the heaps for which the permissions were granted.

Semantic domains

$\ell \in$	Loc		infinite set of locations
$u \in$	Stamp	$=$	$\mathit{Integer}$
$H \in$	Heap	$=$	$\mathit{Loc} \rightharpoonup \mathit{Obj}$
	Obj	$=$	$\mathit{Prop} \rightharpoonup (\mathit{Stamp} \times \mathit{Val})$
$\mathcal{P}, \mathcal{R}, \mathcal{W} \in$			$\mathit{Stamp} \rightharpoonup \mathit{PLang}$
$\mathcal{M}, \mathcal{N} \in$	PMap	$=$	$\mathit{Stamp} \rightharpoonup \mathit{Path}$
$(\ell, \mathcal{M}) \in$	Ref	$=$	$\mathit{Loc} \times \mathit{PMap}$
$v \in$	Val	$=$	$\mathit{Ref} + (\mathit{Env} \times \mathit{Expr})$
$\rho \in$	Env	$=$	$\mathit{Var} \rightharpoonup \mathit{Val}$

Checking permissions

CHECK PERMISSION
$$\frac{\forall u \in \mathrm{dom}(\mathcal{P}) \cap \mathrm{dom}(\mathcal{M}) : \mathcal{M}(u) \in \mathcal{P}(u)}{\mathcal{P} \vdash_{\mathrm{chk}} \mathcal{M}}$$

Evaluation rules

VAR
$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; x \hookrightarrow H; u; \rho(x)$$

LAM
$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \lambda x.e \hookrightarrow H; u; (\rho\downarrow_{FV(\lambda x.e)}, \lambda x.e)$$

APP
$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \hookrightarrow H''; u''; v_1 \\ \rho'[x \mapsto v_1], \mathcal{R}, \mathcal{W} \vdash H''; u''; e \hookrightarrow H'''; u'''; v\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \hookrightarrow H'''; u'''; v}$$

NEW
$$\frac{\ell \notin \mathit{dom}(H)}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \mathtt{new} \hookrightarrow H[\ell \mapsto \emptyset]; u; (\ell, \emptyset)}$$

PUT
$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \\ \mathcal{W} \vdash_{\mathrm{chk}} \mathcal{M}.p \qquad H''' = H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \hookrightarrow H'''; u''+1; v}$$

GET
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \vdash_{\mathrm{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow H'; u'; \mathcal{M}.p \otimes H'(\ell)(p)}$$

PERMIT
$$\frac{\begin{array}{c}\rho', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H; u+1; e \hookrightarrow H'; u'; v \\ \rho' = \rho[x \mapsto \rho(x) \lhd [u \mapsto \varepsilon]]\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \mathtt{permit}\, x : L_r, L_w\, \mathtt{in}\, e \hookrightarrow H'; u'; v}$$

Figure 3. Semantics.

The time stamp of a permission uniquely identifies different executions of permit expressions and determines their relative order with respect to heap modifications.

A value $v \in \mathit{Val}$ is either a reference or a closure consisting of an environment and a lambda expression. The representation of a reference is a pair of a heap address ℓ and a collection $\mathcal{M}$ of access paths, indexed by time stamps. The collection $\mathcal{M}$ records all permitted access paths that have been traversed during evaluation so far to obtain this reference value. The indexing is again used for marking modifications with time stamps. This representation is dictated by the choice of a path-based semantics (see Sec. 2.1).

A heap maps a location to an object and an object maps a property name to a pair of a time stamp and a value. The time stamp indicates the time of the write operation that last assigned the

$$\mathcal{M}' \otimes (u,v) \quad := \quad \begin{cases} (\ell', \mathcal{M}' \otimes_u \mathcal{N}) & \text{if } v = (\ell', \mathcal{N}) \\ v & \text{if } v \notin \mathit{Ref} \end{cases}$$

$$(\mathcal{M} \otimes_u \mathcal{N})(u') \quad := \quad \begin{cases} \mathcal{N}(u') & \text{if } u' \in \mathit{dom}(\mathcal{N}) \\ \mathcal{M}(u') & \text{if } u' \in \mathit{dom}(\mathcal{M}) \backslash \mathit{dom}(\mathcal{N}) \wedge u < u' \\ \text{undefined} & \text{if } u' \in \mathit{dom}(\mathcal{M}) \backslash \mathit{dom}(\mathcal{N}) \wedge u \geq u' \\ \text{undefined} & \text{if } u' \notin \mathit{dom}(\mathcal{M}) \cup \mathit{dom}(\mathcal{N}) \end{cases}$$

$$(\mathcal{M}.p)(u) \quad := \quad \begin{cases} \mathcal{M}(u).p & \text{if } u \in \mathit{dom}(\mathcal{M}) \\ \text{undefined} & \text{if } u \notin \mathit{dom}(\mathcal{M}) \end{cases}$$

$$v \lhd \mathcal{M} \quad := \quad \begin{cases} (\ell, \mathcal{N} \lhd \mathcal{M}) & \text{if } v = (\ell, \mathcal{N}) \\ v & \text{if } v \notin \mathit{Ref} \end{cases}$$

$$(\mathcal{N} \lhd \mathcal{M})(u) \quad := \quad \begin{cases} \mathcal{M}(u) & \text{if } u \in \mathit{dom}(\mathcal{M}) \\ \mathcal{N}(u) & \text{if } u \notin \mathit{dom}(\mathcal{M}) \end{cases}$$

Figure 4. Auxiliary definitions.

property. It is required to implement the "last writer wins" principle from Sec. 2.4.

The evaluation rules VAR, LAM, and APP for variable, lambda abstraction, and function application expressions are standard. They thread the time stamp and propagate the permissions $\mathcal{R}$ and $\mathcal{W}$ unchanged to their sub-evaluations, if any.

The evaluation rule NEW creates a new object in the heap. This object has no properties and its collection of access paths is empty. The latter indicates that the newly created object is completely unrestricted (following Sec. 2.3). Any of its properties may be read or written. The time stamp does not change when allocating a new object because no object in the current heap is modified.

The PUT rule specifies the operation that writes and (if necessary) defines a property. It first computes the location ℓ and the collection $\mathcal{M}$ of access paths of the object and then checks the write permission to the object with the premise $\mathcal{W} \vdash_{\text{chk}} \mathcal{M}.p$. It overwrites the object's property with the new value and assigns it a new, incremented time stamp to implement the "last writer wins" principle (Sec. 2.4). Hence, the time stamp of a property is always the time of its last update.

The rule GET defines the read operation of object properties. It relies on some auxiliary operations defined in Figure 4. It first expects e to evaluate to a reference $(\ell, \mathcal{M})$, which denotes the base object for the property read. In this reference, ℓ is the heap address of the object and $\mathcal{M}$ contains a collection of access paths for the object corresponding to heap traversals reaching this object, one path for each active access permission. The other premise $\mathcal{R} \vdash_{\text{chk}} \mathcal{M}.p$ checks the read permission for these paths extended with property p. This check is specified by rule CHECK PERMISSION which requires that, for each active access permission with time stamp u, the current access path for u is an element of the set of permitted access paths for u, i.e., $\mathcal{P}(u)$.

If the read operation is permitted, then there are two possibilities. If the property contains a closure, then this closure is the result of $e.p$. However, if the property contains an object reference, say $(\ell', \mathcal{N})$, then this read operation has discovered that $\mathcal{M}.p$ are also access paths for object ℓ'. The reference value returned from the read operation must somehow merge the different ways to reach ℓ': via $\mathcal{N}$ and via $\mathcal{M}.p$. Computing the desired collection of access paths depends on the last time u when the property $\ell.p$ was updated. This complication is required to implement the pre-state snapshot principle (Sec. 2.3).

The operator $\otimes$ in Fig. 4 implements the required merger operation. Its right-hand argument are the contents of an object's property: (u, v) where u is the time stamp of the last update and v the stored value. Its left-hand argument is the collection $\mathcal{M}' = \mathcal{M}.p$ of newly discovered paths to the property. If v is not a reference, $\otimes$ just returns v as already discussed. Otherwise, $v = (\ell', \mathcal{N})$ in which case it returns the location ℓ' paired with the collection of paths computed by the operator $\otimes_u$ applied to the new paths $\mathcal{M}'$ and the old paths $\mathcal{N}$.

In an application $\mathcal{M} \otimes_u \mathcal{N}$, the first argument $\mathcal{M}$ contains the access paths that were detected when checking the read access. The second argument $\mathcal{N}$ contains the access paths as they are stored in the heap at location ℓ'. The subscript u is the time stamp of the last write to the property. The definition in Figure 4 distinguishes three cases depending on when the property has been written last and what access paths were given to the written value. Let u' be the time stamp of an execution of a permit expression.

1. The object's property value already has an access path for index u' (in $\mathcal{N}$). Thus, the property has been overwritten after the installation of u'. In this case, any new access path in $\mathcal{M}$ is ignored. Instead, the existing access path is returned according to the pre-state snapshot principle (Sec. 2.3) as it reflects an access path at the time when the permission attached to u' has been installed.
2. The object's property value has no access path for index u' (in $\mathcal{N}$) and it had been written *before* the permission with index u' has been installed as can be seen from $u < u'$. In this case, we attach the new u'-path to the value. This path is realizable in the pre-state snapshot at time u' because the property has been written at $u < u'$, that is, before u'.
3. There is no access path for index u' (in $\mathcal{N}$) and the property has been written *after* the contract with index u' has been installed (viz. $u \geq u'$). In this case, no u'-path is attached because this property was not linked to the data structure in the pre-state snapshot at time u'.

The examples in Section 3.3 illustrate these three cases.

The rule PERMIT installs an access permission contract. Each such permission is bound to the time stamp u of the heap in which the permission is installed. It increments the time stamp to avoid clashes with the next permission. Then, evaluation proceeds with the body of the permit-expression, but with an updated variable binding for x, which records the time stamp u for the heap reachable from the object bound to x (if any) by attaching $[u \mapsto \varepsilon]$ to it, and updated read and write permissions, which record the stated permission set L_r and L_w for the object network reachable from x.

An access permission has dynamic extent (Sec. 2.2) because the access permissions are propagated with the flow of execution and the rule CHECK PERMISSION only considers the entry points in the domain of the current access permission $\mathcal{P}$. In particular, access permission contracts are not captured by closures created while they are in force: Closure creation (rule LAM) ignores the access permissions and function application (rule APP) continues to use the current permissions with the body of the invoked function. Hence, after evaluation of the body of an access permission is complete, the information associated with its index u could be garbage collected both from the value and from the heap.

3.3 Examples

The code fragments in Fig. 5 illustrate the different cases of the $\otimes_u$ operator. The fragments (a) and (b) correspond to the examples l1 and m1 in Sec. 2.4. They differ only in the placement of the permit expression. The code fragment (a) installs the permission *before* creating an alias with the assignment x.a = x.b whereas version

```
1 let x = new in
2   x.a = new;
3   x.b = new;
4   permit x :
5     {a,b,b.a},{a,b.a} in
6     x.a = x.b;
7     x.a.a = 42
```

(a) Valid access

```
1 let x = new in
2   x.a = new;
3   x.b = new;
4   x.a = x.b;
5   permit x :
6     {a,b,b.a},{a,b.a} in
7     x.a.a = 42
```

(b) Invalid access

```
1 let x = new in
2 let y = new in
3   x.a = new;
4   permit y : {a}, {a} in
5   permit x : {a}, {a} in
6     x.a = y;
7     x.a.a = 42
```

(c) Nested permissions

Figure 5. Exercising the definition of $\otimes$.

PERMIT'

$$\frac{\begin{array}{c}\rho', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w], \mathcal{F}' \vdash' H; u+1; e \hookrightarrow H'; u'; v \\ \rho' = \rho[x \mapsto \rho(x) \lhd [u \mapsto \varepsilon]] \\ \mathcal{F}' = \text{ if } \rho(x) = (\ell, \mathcal{M}) \text{ then } \mathcal{F}[u \mapsto (\ell, H)] \text{ else } \mathcal{F}\end{array}}{\rho, \mathcal{R}, \mathcal{W}, \mathcal{F} \vdash' H; u; \texttt{permit}\, x : L_r, L_w \,\texttt{in}\, e \hookrightarrow H'; u'; v}$$

Figure 6. Gathering foretime information.

(b) installs the permission afterwards. In both cases, let the permit expression be associated with time stamp u'.

In fragment (a), the expression x.b in line 6 returns the location ℓ_b paired with the map $[u' \mapsto b]$ (case 2 of $\otimes_u$: $u < u'$ because it was generated by the preceding assignment x.b = **new**). This value is written to x.a. The following access to x.a returns $(\ell_b, [u' \mapsto b])$ according to case 1 of $\otimes_u$ which governs that the paths stored in the object take precedence over the actual path taken. For the final write access, the extended access map $[u' \mapsto b.a]$ is checked against the set of write permissions and succeeds.

In fragment (b), x.a = x.b is executed before the permit expression. Hence, x.a contains $(\ell_b, \emptyset)$ and the GET rule makes it return $(\ell_b, [u' \mapsto a])$ according to case 1 of $\otimes_u$. For the write operation, the extended access map $[u' \mapsto a.a]$ is checked against the set of write permissions and fails.

The code in Figure 5(c) exercises case 3 of the definition of $\otimes_u$. After establishing the two permissions, the environment ρ is: $[x \mapsto (\ell_x, [u_3 \mapsto \varepsilon]), y \mapsto (\ell_y, [u_2 \mapsto \varepsilon])]$ where the u_i are sorted according to their indexes i. After the assignment x.a = y (with time stamp u_4) the object in location ℓ_x is: $\{a : (u_4, (\ell_y, [u_2 \mapsto \varepsilon]))\}$. In line 7, x.a evaluates to

$$\begin{array}{ll} & [u_3 \mapsto a] \otimes (u_4, (\ell_y, [u_2 \mapsto \varepsilon])) \\ = & (\ell_y, [u_3 \mapsto a] \otimes_{u_4} [u_2 \mapsto \varepsilon]) \\ = & (\ell_y, [u_2 \mapsto \varepsilon]) \end{array}$$

Observe that case 3 of $\otimes_u$ applies because $u_4 \geq u_3$. In consequence, u_3 vanishes from the domain of the map because the object that was reachable via x.a before line 6 has become garbage. With this reasoning the update of x.a.a is permitted because it is equivalent to y.a and realizable in the heap after line 5.

3.4 Pre-state Snapshot

The first result is a soundness results that underlines that our semantics adheres to the pre-state snapshot principle (Sec. 2.3). Basically, we want any value produced during evaluation to contain correct path information with respect to all relevant pre-states in the following sense. Consider a reference value of the form $(\ell, \mathcal{M})$ where $\mathcal{M}$ contains a map from time stamps to access paths. Each time stamp $u \in \mathit{dom}(\mathcal{M})$ indicates an installed access permission that affects this reference value. That is, for each time stamp $u \in \mathit{dom}(\mathcal{M})$ there is an access permission installed at time u for heap H_u with base object ℓ_u. The information contained in $\mathcal{M}$ is correct with respect to such u if there is a path from the base object ℓ_u to ℓ along the properties of $\mathcal{M}(u)$ in the pre-state heap H_u.

To formally define this notion, suppose the information about the pre-states and the base objects of all contract installations is gathered in a time-stamp indexed *foretime map* $\mathcal{F} : \mathit{Stamp} \rightharpoonup (\mathit{Loc} \times \mathit{Heap})$. It maps the time stamp u of the installation of an access permission to a pair (ℓ_u, H_u), where ℓ_u is the location of the base object and H_u is the heap snapshot at that time.

Definition 3.1 *Let $\mathcal{F}$ be a foretime map.*
A value v is $\mathcal{F}$ path consistent *if*

- *$v = (\rho, \lambda x.e)$ and ρ is $\mathcal{F}$ path consistent or*
- *$v = (\ell, \mathcal{M})$ and, for all $u \in \mathit{dom}(\mathcal{M})$, if $\mathcal{F}(u) = (\ell_u, H_u)$, then there is a path from ℓ_u to ℓ along $\mathcal{M}(u)$ in H_u.*

An environment ρ is $\mathcal{F}$ path consistent *if, for all $x \in \mathit{dom}(\rho)$, $\rho(x)$ is $\mathcal{F}$ path consistent.*

A heap H is $\mathcal{F}$ path consistent if all values stored in all object properties are $\mathcal{F}$ path consistent. That is, for all $\ell \in \mathit{dom}(H)$ and for all $p \in \mathit{dom}(H(\ell))$, if $H(\ell)(p) = (u, v)$, then v is $\mathcal{F}$ path consistent.

To gather the foretime map, a suitably extended evaluation judgment $\rho, \mathcal{R}, \mathcal{W}, \mathcal{F} \vdash' H; u; e \hookrightarrow H'; u'; v$ is required. It records the base object and the heap snapshot at each successful contract installation in the foretime map $\mathcal{F}$. Fig. 6 contains the correspondingly modified PERMIT' rule. The remaining rules for the extended judgment extend the ones for the original judgment in Fig. 3 by passing the foretime map in exactly the same way as $\mathcal{R}$ and $\mathcal{W}$.

Showing adherence to the pre-state snapshot principle amounts to proving that an evaluation that starts on a path consistent heap and environment produces a path consistent heap and value.

Theorem 3.1 *Suppose that $\rho, \mathcal{R}, \mathcal{W}, \mathcal{F} \vdash' H; u; e \hookrightarrow H'; u'; v$.*
If ρ and H are $\mathcal{F}$ path consistent, then so are H' and v.

We also proved an accompanying completeness result that guarantees that the $\mathcal{M}$ component of a reference value is non-empty if it has been accessed via a pre-state path.[5]

3.5 Stability of Violation

Stability of violation is a property linked to the reference attachment principle (Sec. 2.1). It states that a violation of an access permission is preserved (in a precisely defined sense) when performing the same computation on a heap with more aliasing.

Let's first fix what we mean with "more aliasing." If H_1 and H_2 are heaps, then H_2 has more aliasing if it identifies locations that are distinct in H_1 and merges the contents of the objects in these locations. That is, if o' and o'' are distinct objects in H_1 which are merged to object o in H_2, then o has all properties from o' and o''. Properties present in o' and o'' must have suitably related values that map into the same value in H_2. We call H_1 a *refinement* of H_2 because it makes more distinctions between objects.

Definition 3.2 *A heap H_1 is a γ*-refinement *of heap H_2, written as $H_1 \succcurlyeq_\gamma H_2$, if $\gamma : \mathit{dom}(H_1) \rightarrow \mathit{dom}(H_2)$ is a surjective mapping between heap locations and $\forall \ell_1 \in \mathit{dom}(H_1)$, $o_1 = H_1(\ell_1)$, $o_2 = H_2(\gamma(\ell_1))$:*

[5] See appendix.

GET-CRASH2

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \not\vdash_{\mathrm{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^{R}}$$

GET-CRASH3

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \vdash_{\mathrm{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^{O}}$$

PUT-CRASH3

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \qquad \mathcal{W} \not\vdash_{\mathrm{chk}} \mathcal{M}.p\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^{W}}$$

Figure 7. Essential crashing rules.

RH1 *$dom(o_1) \subseteq dom(o_2)$ (objects in the refined heap have fewer properties) and*

RH2 $(\forall p \in dom(o_1))\ o_1(p) = (u_1, v_1) \wedge o_2(p) = (u_2, v_2) \wedge u_1 = u_2 \Rightarrow v_1 \succcurlyeq_\gamma v_2$

A value is a γ-refinement of another, $v_1 \succcurlyeq_\gamma v_2$ iff

RV1 *$v_1 = (\ell_1, \mathcal{M}_1)$ and $v_2 = (\ell_2, \mathcal{M}_2)$ and $\ell_2 = \gamma(\ell_1)$ and $\mathcal{M}_1 = \mathcal{M}_2$, or*

RV2 *$v_1 = (\rho_1, e_1)$ and $v_2 = (\rho_2, e_2)$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and $e_1 = e_2$.*

An environment is a γ-refinement of another, $\rho_1 \succcurlyeq_\gamma \rho_2$ iff

RE1 *$dom(\rho_1) = dom(\rho_2)$ and*

RE2 $(\forall x \in dom(\rho_1))\ \rho_1(x) \succcurlyeq_\gamma \rho_2(x)$.

The reader might wonder about the implication in **RH2**. This choice allows the coarser heap H_2 to contain a value which does not refine to all corresponding values in heap H_1: for each object in H_2, there may be any number of γ-preimages of this object in H_1. **RH2** says that such an object need not be consistent with all its preimages. This case can be detected by the condition $u_1 < u_2$: the shared version of the object has been updated after one of its unshared preimages. The remaining case $u_1 > u_2$ can never arise.

We allow such inconsistencies in a heap refinement because they only influence the semantics of an program if there is a subsequent read operation that observes the inconsistency. In this case, the criterion $u_1 < u_2$ detects the inconsistency.

Having established the notion of heap refinement, it remains to formalize running the same program on two heaps and compare the outcomes. To this end, it is not sufficient to consider successful, terminating evaluations, but also evaluations ending in a contract violation and interrupted evaluations. Figure 7 specifies the key rules of three judgments of the form $\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \Uparrow^i$ where $i \in \{R, W, O\}$. Each judgment formalizes an interrupted evaluation. The superscript R indicates violation of a read permission (rule GET-CRASH2), superscript W indicates violation of a write permission (rule PUT-CRASH3), and superscript O indicates nondeterministically giving up on a read operation (rule GET-CRASH3). The remaining rules[6] are straightforward variants of the evaluation rules in Fig. 3 that propagate an error condition like an exception.

Our theorem says essentially that crashes due to violated read or write permissions are preserved when more aliasing is added. The main complication is that an inconsistent read operation (in the sense discussed after Definition 3.2) in the version with additional aliasing may lead to arbitrary behavior of the program, including non-termination. Therefore, the theorem constructs a related execution up to the first inconsistent read. Its proof along with auxiliary definitions may be found in the appendix.

[6] See appendix.

$[\![\ e_1[e_2]\ ]\!]$	=	pRead($[\![e_1]\!]$,$[\![e_2]\!]$)
$[\![\ e_1[e_2] = e_3\ ]\!]$	=	pAssign($[\![e_1]\!]$,$[\![e_2]\!]$,$[\![e_3]\!]$)
$[\![\ e(e_1,\ldots,e_n)\ ]\!]$	=	fCall($[\![e]\!]$,[$[\![e_1]\!]$,...,$[\![e_n]\!]$])
$[\![\ e.$m$(e_1,\ldots,e_n)\ ]\!]$	=	mCall($[\![e]\!]$,m,[$[\![e_1]\!]$,...,$[\![e_n]\!]$])
$[\![$ **new** $e(e_1,\ldots,e_n)\ ]\!]$	=	cCall($[\![e]\!]$,[$[\![e_1]\!]$,...,$[\![e_n]\!]$])
$[\![$ **for** (**var** i **in** e) $\{s\}\ ]\!]$	=	**var** o=$[\![e]\!]$; **for** (**var** i **in** o}) { **if** (mCall(o,"hoP",[i])) {$[\![s]\!]$}}
$[\![$ **function** $f(x,\ldots)$ $\{s\}\ ]\!]$	=	**var** f = enableWrapper(**function** f'$(x,\ldots)$ {$[\![s]\!]$})

Figure 8. Transformation rules (excerpt).

Theorem 3.2 *If $H_1 \succcurlyeq_\gamma H_2$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and*

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \Uparrow^i \tag{1}$$

(for $i \in \{R, W\}$) then

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \Uparrow^j \tag{2}$$

such that either $i = j$ or $j = O$ and the derivation of (2) ends in an inconsistent read operation with respect to (1).

Informally, the proof constructs a derivation of (2) from a derivation of (1). As $H_1 \succcurlyeq_\gamma H_2$, it is either the case that (2) always reads the same values from the heap as (1). In this case, the derivation of (2) is isomorphic to the derivation of (1) and $i = j$. Otherwise, there is an instance of a GET rule in the derivation of (1) such that applying this rule as part of (2) would return a different value. A sufficient criterion for this case is to check if $u_1 < u_2$ when reading the property as shown in **RH2**. In this case, we give up, emit a GET-CRASH3 rule instead of GET, and complete the derivation with propagation rules for $\Uparrow^O$.

The theorem also holds in a language with conditionals as they can be simulated in the lambda calculus. If the language were extended with pointer equality, then a condition might turn out differently on H_2 than on H_1. However, Theorem 3.2 would still hold if a rule analogous to GET-CRASH3 were introduced that allowed us to derive a $\Uparrow^O$ judgment in (2) instead of executing an inconsistent pointer equality.

4. Implementation

The implementation of the framework for monitoring access permissions consists of two parts. The first part is an off-line JavaScript-to-JavaScript compiler written in OCaml. The second part is a JavaScript library that handles the dynamic aspects of enforcing access permissions. Both are available from our webpage.[7]

The implementation supports the full JavaScript language according to the standard [11]. As in the examples in Sec. 2, contracts can be attached to a function or method in a special kind of comment /*c ... */.

The compiler transforms the annotated code such that it monitors access permissions at run time. Figure 8 illustrates some of the transformation steps in simplified form. All operations that involve heap accesses, like reading and writing of properties, are redirected to library functions that dynamically manage access permissions. These library functions introduce wrappers for references that remember the access paths used to reach the wrapped reference.

Figure 9 shows an example of a transformed function definition. The library function enableWrapper creates a wrapper for f that generates a fresh time stamp each time a function is called and marks the parameters with the corresponding access path information at run time such that pRead can check if it has permission to read $1.a. The library call to pRead returns a wrapper with the access path $1.a for z. Reading the property b of z uses the access

[7] Reference submitted with paper.

```
/** (any) → any
    with [ $1.a.b ] */
function f(x) {
  var z = x.a;
  return z.b;
};
```

⇝

```
TESTS.c1 = ... // contract
var f = enableWrapper(
  function f'(x) {
    var z = pRead(x,"a");
    return pRead(z,"b");
  }, [TESTS.c1]);
```

Figure 9. Example of a transformation.

path stored in the wrapper of z, extends it to $1.a.b, and checks if reading this path is permitted. The permission is granted because the access permission attached to f is $1.a.b.

Calls to native or non-transformed code would fail if wrapped objects were passed. Because it is not possible to statically decide which function is applied at a call site, the framework strips the access meta data from parameter objects before passing them to the function. It stores the meta data on a global stack that is used to re-wrap the objects if the callee itself is a transformed function. This approach is compatible with uses of eval, although monitoring does not extend to eval-generated code.

For interoperability with non-transformed code, it is also necessary to remove wrappers when storing object properties. To this end, an additional map (__infos__) is attached to each object. This map stores the wrappers for each of the properties. The function pRead uses this map to reconstruct wrapped objects if necessary.

As the library stores the access path information in the __infos__ property of the objects, this property must not become accessible to user code. Therefore, we provide a substitute for hasOwnProperty (hoP) that masks out the __infos__ property. We also transform the statement **for** (**var** i **in** e) { s } to ensure that internal properties used by the implementation do not leak out to the program. Technically, this protection is achieved by changing the body s to **if** (hoP(o,i)) { s }. The functions pRead and pAssign also safeguard the special property __infos__.

If native code or non-transformed code iterates over all properties of an object, then it is not possible to hide the __infos__ property. We are not aware of any way to reliably hide this property short of modifying the underlying JavaScript engine. However, in the case studies that we performed the special property caused no problem.

5. Evaluation

How effective are access permissions for detecting programming errors? To answer this question, we hand-annotated the code of several libraries and applications with contracts and ran it with monitoring enabled. We applied random code modifications [7] to check to what extent the enforcement of access permissions detects changes in the program's behavior.

5.1 Case Study: Singly- and Doubly-Linked Lists and Trees

The first case study examines a collection of libraries implementing data structures like singly- and doubly-linked lists and search trees.[8] The code sizes range from 200-400 LOC per library. Because the results are similar for all libraries, we discuss the results for the singly-linked list implementation as a representative example.

The list interface comprises one constructor for list nodes and six methods to operate on a list: add, remove, item, size, toArray, and toString. For each method we developed contracts with access permissions. Annotating the code and implementing a custom contract to drive the input generation took about one hour. The code with all contracts is available on our webpage.

	type		type+effect	
fulfilled contracts	1011	18.0 %	711	12.7 %
rejected contracts	4607	82.0 %	4907	87.3 %
reason for rejection (a mutant may be counted multiple times)				
type contract failure	2020	43.9 %	1643	33.5 %
signaled error	2034	44.1 %	2136	43.5 %
browser timeout	553	12.0 %	243	5.0 %
read violation	-	0.0 %	1018	20.7 %
write violation	-	0.0 %	1606	32.7 %
read/write violation	-	0.0 %	1842	37.5 %

Table 1. Testing random mutations for singly-linked lists.

From the implementation we derived about 5600 random mutations and tested each mutant against the original contracts. The mutations affected operators, constants, and variable names. Each of the six functions was tested with 1000 randomly generated test cases and was run in two configurations:

type contracts specifying integer lists without effects: only violations of the type contracts are detected,

type+effect contracts specifying integer lists with effects: type and access path violations are detected.

Table 1 shows the results of the test runs. The "fulfilled" row counts mutations that are not detected because the mutant fulfills all six contracts. The "rejected" row registers mutants that fail at least one contract. These two rows indicate the effectiveness of effect monitoring. Adding access permissions to type contracts improves the detection rate for mutations from 82% to 87.3%, an improvement of 6.4%. The remaining rows break down the reasons for the failure of a mutant. As there are multiple functions in a mutant, there are multiple reasons why a single mutant may fail so that the percentages do not add to 100%.

We manually inspected the cases where the contract system did not detect a mutation. In many cases the mutated code is semantically equivalent to the original version, for instance, when x.p was changed to x.q, where both properties p and q were always undefined. In other cases, the contract was fulfilled by a mutant because the modification did not change any property access or return value from a type perspective, for instance, **return** true is changed to **return** false. While these mutations changed the semantics, a type or access permission contract cannot detect such changes.

We also manually inspected ten randomly selected mutants that timed out. In all cases, the mutation caused an infinite loop.

5.2 Case Study: Richards and Deltablue Benchmarks

A second case study was performed on the Richards and Deltablue benchmarks from the Google V8 benchmark suite. The Richards benchmark[9] simulates the task dispatcher of an operating system. The code comprises 29 functions in 650 LOC. A person without prior knowledge of the code under test provided the contracts and implemented custom generators in about four hours. It took another two hours to develop the access permissions.

Table 3 shows the result of testing about 2950 mutated versions. These test runs executed 50 tests per function for each mutant to test effect and contract violations. We chose to run a smaller number of tests to reduce the overall run time and to check if a small number of test cases is sufficient to obtain a high detection rate of mutants.

For this application, adding access permissions increased the detection rate from 61.1% to 69.2%, which amounts to a 13% improvement. This increase is quite surprising as the percentages

[8] https://github.com/nzakas/computer-science-in-javascript

[9] http://v8.googlecode.com/svn/data/benchmarks/v6/richards.js

	type		type + effect	
fulfilled contracts	1148	38.9%	911	30.8%
rejected contracts	1807	61.1%	2044	69.2%
reason for rejection (a mutant may be counted multiple times)				
type contract failure	872	48.3%	866	42.4%
signaled error	1052	58.2%	1037	50.7%
browser timeout	28	1.5%	30	1.5%
read violation	0	0.0%	202	9.9%
write violation	0	0.0%	149	7.4%
read/write violation	0	0.0%	349	17.1%

Table 2. Testing random mutations of the Richards case study.

of detected read or write violations are much smaller than in the linked-list case study.

In the Deltablue application (59 contracts in 670 LOC), another case study from the V8 benchmark suite, the access permissions led to an increase in error detection from 75.6% to 84.2%, an improvement by 11.4%. In contrast to the other case studies, Deltablue does not permit unit testing on a per function basis as it relies heavily on global state. Further details may be found in the appendix.

5.3 Performance Evaluation

All case studies and benchmarks were executed on a Lenovo Thinkpad X61s notebook with a Core 2 Duo processor with 1.60 GHz and 2GB Ram running the Google-Chrome browser (7.0.517.44) on top of Linux version 2.6.35-23-generic. In this setting, a test run of a mutant is about four times slower with monitoring enabled than without monitoring (in both case studies).

To give ballpark numbers, running 1000 tests for each of the six functions of the linked-list test suite takes about 6 seconds with monitoring compared to 1.5 seconds without. For Richards, running 50 tests for each of the 29 functions takes 1.85 seconds with monitoring compared to 0.5 seconds without.

For the Richards benchmark we timed the original code (without mutation) once with monitoring and once without to measure the slowdown for code that never violates the effect annotations. This experiment masks out the effects of contract violations, which cause the program to stop earlier on faulty mutants than on correct code. However, the slowdown is similar: running 1000 test cases for each of the 29 functions took 32.4 seconds with monitoring enabled versus 7.4 seconds without, a slowdown factor of 4.4.

6. Application to Security

A typical attacker model for JavaScript considers untrustworthy code that is loaded at run time to enhance one's program with some sought-after functionality. The lack of a proper module system or encapsulation mechanism in JavaScript means that the attacker's code can arbitrarily explore and modify any accessible object. This freedom is undesirable because the code may leak or change sensitive information contained in some of the objects, for example, the browser history, session cookies, the currently displayed page, inputs into a web form, and so on.

Access permission contracts with the path-based semantics can ensure that certain parts of the object graph are never accessed after extending the monitoring framework as follows: When starting a script, an implicit permission is installed for the roots of all sensitive information. Typically, we would install the permission window.∗, which enables read and write access to all transitive properties of the window object. Effectively, this permission covers all objects accessible to a JavaScript program.

The novelty of the extension is that the installation of this permission returns a handle, say window_handle, that allows us to later restrict the permission by removing access paths from it. For example, we would invoke an untrusted function through a permission wrapper like the following to keep it from changing the location property of the window object.

```
/*c () → any with [...] except [window_handle.location] */
function untrusted_wrapper() {
  untrusted();
}
```

This way, a program can explicitly manage the trust invested in a foreign code fragment.

The installation of such an exception would still take constant time, as the implementation only needs to enter the exception into the permission map for window_handle. Also the cost of checking a permission remains linear in the number of installed permissions.

In the formal system, the `permit` expression would change to $\texttt{permit}\,x\ \texttt{as}\ y : L_r, L_w\ \texttt{in}\,e$ where the execution additionally binds y to the time stamp under which the permission for x is installed. A new expression $\texttt{restrict}\ y : L_r, L_w\ \texttt{in}\ e$ removes the permissions L_r and L_w from $\mathcal{R}(u)$ and $\mathcal{W}(u)$ when y is bound to u. Details may be found in the appendix.

Our implementation already supports the **except** clause with a slightly different semantics, but it does not provide permission handles, yet. We are currently working on this addition as a proof of concept, but in the long run we aim for a browser-based implementation. The transformation-based implementation is sufficient for the applications related to program understanding and testing, but for the security application, the implementation must take place inside the browser. A browser-based implementation is less brittle than a transformation-based implementation, it imposes less runtime overhead on the program execution, and it is able to fully deal with the dynamicity of JavaScript, in particular with the frequent uses of eval and with the dynamic loading of parts of the program in various ways.

Extending the location-based semantics for security applications also requires the introduction of an **except** clause. Processing such a clause would give rise to yet another traversal of the object graph, but this time removing permissions rather than joining them.

7. Related Work

Access permissions are closely related to effect systems. Effect systems have been conceived for functional languages [19] to describe and infer the scope of side effects, with the goal of detecting parallelizable code fragments and improving memory management.

There are too many papers on effect systems to do them all justice here. Greenhouse and Boyland [22] introduce effect annotations for Java which closely resemble our contracts. In contrast to our system, effects are collected for regions which comprise a set of objects. Their approach aims to track data dependencies of software components. The main differences to our work are that most effect systems are integrated in type systems and thus geared towards static analysis (whereas ours performs dynamic analysis) and that our prime motivation lies in the detection of software defects.

Interestingly, the effect system proposed by Bocchino and coworkers [28] for deterministic parallel Java relies on a very similar notion of effect, based on paths over regions (sets of instance variables). They use the effect system to statically prove the absence of data races. Our system might be extended to check this property at run time.

Similarly related is work on ownership and aliasing control. Again, with the exception of the dynamic ownership system of Boyland and coworkers [6], most ownership systems statically impose tree-like ownership structures on object graphs [3, 10, 39, 45]. The main difference to ownership types is that our system is entirely access path-based whereas ownership types are context-based. Furthermore, some ownership systems forbid the mere existence of references, whereas access permissions forbid the traversal of certain paths. Effective ownership [33] does not restrict referencing of objects, but enforces the encapsulation of an object's representation by confining modifications to the owner.

Bierhoff and Aldrich [5] define a static checker for access permissions in Java. It combines typestate and object aliasing information to design and verify protocols for safe object access. They also focus on the correct usage of single resources. Their access permissions are statically verified.

Deutsch's [8] analysis for sharing and aliasing is also entirely based on access paths. It is a static analysis phrased as an abstract interpretation of a storeless semantics.

Run-time monitoring is an approach to providing safety and security guarantees. Erlingsson [13] provides an overview of such applications. As a notable difference, security monitoring is mostly geared towards eliminating (sequences of) uses of undesired operations and can often be implemented by finite automata, whereas access path monitoring rules out undesired accesses and requires more specific implementation techniques to deal with aliasing.

BrowserShield [40] provides run-time monitoring of JavaScript. BrowserShield rewrites code to redirect critical operations according to user-specified policies. The Google Caja project [20] employs an online compilation process of JavaScript code to a safe subset named Cajita which enforces certain security policies.

Maffeis and coworkers [34] combine several isolation techniques for restricting heap accesses of third-party code. They disallow eval, **function**, and constructor within untrusted code and also rewrite property accesses with wrappers to enable run-time checks.

These systems operate within the browser during interactive user sessions and provide complete interposition. In contrast, our tool is focused on development and testing of applications.

Finifter and coworkers [18] design a JavaScript heap analysis framework to detect information leaks. To prevent exploits, third-party code is restricted to a name space by prefixing properties with a unique identifier. In contrast, we restrict accesses via path conditions.

ConScript [37] enforces fine-grained application-specific security policies at run time by a modified JavaScript execution engine. Compared to our approach, they have different goals and less overhead, but are tied to a particular browser implementation.

Further related research deals with dynamic contract checking. Findler and Felleisen [16] develop a dynamically checked type contracts for Scheme. In a similar way, JSConTest [24] extends JavaScript with type contracts that are monitored dynamically and can be used to automatically generate random test cases for contracted functions. Our paper extends their work with access permissions to check side effects of a contracted function.

Program specification frameworks like Spec# [4] or Eiffel [12] permit the formulation of access permissions as FOL-formulas in Hoare-style pre- and postconditions. Because specialized syntax is missing, the annotation process is rather heavy-weight. Besides, these frameworks are geared towards full specifications, whereas we are targeting partial specifications.

JML [31] features an assignable clause to specify which object fields may be modified during a method call, similar to our write permissions (read accesses cannot be restricted). In contrast to our work, JML is mostly geared towards (static) program verification. Of the approaches that employ JML for run-time checks (that is, contract monitoring) only a few fully support assignable. Lehner and Müller [32] provide such an implementation of run-time checks. Their implementation relies on code rewriting, but the main contribution of their work is to efficiently check assignable clauses for dynamic data groups. The semantics of these checks is reminiscent of the location-based semantics discussed in Sec. 1.3.

Spoto and Poll [41] define a static analysis for object-local assignable specifications. They include alias information in their system by tracking what aliases are introduced when a field is modified. Their analysis also seems to implement the location-based semantics.

8. Conclusion

We proposed a novel extension of software contracts with access permissions that specify the side effects of an operation in terms of access paths. We implemented monitoring for access permissions in JavaScript by a program transformation and demonstrated that this implementation has an acceptable overhead. As a basis of the implementation, we developed a formalization that enabled us to cleanly specify the interaction of monitoring and aliasing, to prove soundness of monitoring, and to prove stability of violation.

Our case studies showed that the specification of contracts with access permissions on an unfamiliar code base takes about 30-40 minutes per 100 LOC and that in return the number of bugs detected by contract monitoring increases between 6% and 13%, which is a remarkable improvement on type contracts. In each case, the access permissions provided valuable insights in the behavior of the program. Hence, access permissions could be a worthwhile extension of testing frameworks.

8.1 Effect Inference

For program understanding and regression testing, it is advantageous to automatically infer access permission contracts as their manual construction can be tiresome and error prone. To this end, we defined and implemented a heuristic algorithm[26] which infers access permissions from the set of access paths that are collected during several test runs. Our experiments (which include the code discussed in Sec. 5.1 and 5.2) showed that effect contracts of high precision can be inferred with only minimal manual interaction. We take this result as additional evidence that access permission contracts can play an important role in automatic regression testing.

8.2 Future Work

In future work, we want to pursue various directions. We believe that our approach is more widely applicable to any object-based language, not just to scripting languages, but also to nominally typed languages like Java and C#. In the latter cases, to avoid breaking encapsulation, it is likely necessary to introduce a concept like regions or datagroups analogous to other work in this area [22, 28, 31]. In this context, it would also be interesting to investigate a mix of static checking and dynamic enforcement as in work on manifest contracts [21].

Another obvious extension would be a special treatment for effects on the DOM [30]. Because DOM structures are guaranteed to be trees (no aliasing!), many of the complications of general object graphs do not arise in the case of DOM structures.

The extensions to reliably support the security application outlined in Sec. 6 seem very promising. We are currently working on a browser-based implementation of access permission monitoring to further validate this application.

References

[1] P. Abercrombie and M. Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proc. 38th ACM Symp. POPL*, pages 201–214, Austin, USA, Jan. 2011. ACM Press.

[3] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, pages 1–25, Oslo, Norway, June 2004. Springer.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69. Springer, 2004.

[5] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. 22nd ACM Conf. OOPSLA*, pages 301–320, Montreal, QC, CA, 2007. ACM Press, New York.

[6] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proc. 15th European Conference on Object-Oriented Programming*, pages 2–27, London, UK, 2001. Springer-Verlag.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[8] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. IEEE International Conference on Computer Languages 1992*, pages 2–13, Oakland, CA, Apr. 1992. IEEE.

[9] T. D'Hondt, editor. *24th ECOOP*, volume 6183 of *LNCS*, Maribor, Slovenia, 2010. Springer.

[10] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, Oct. 2005.

[11] ECMAScript Language Specification, Dec. 2009. ECMA International, ECMA-262, 5th edition.

[12] Eiffel: Analysis, design and programming language, June 2006. ECMA International, ECMA-367, 2nd edition.

[13] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, Sept. 1999.

[14] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 2103–2110, Sierre, Switzerland, 2010. ACM.

[15] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.

[16] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

[17] R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In O. Chitil, Z. Horváth, and V. Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007*, number 5083 in Lecture Notes in Computer Science, pages 111–128. Springer, 2008.

[18] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of Network and Distributed System Security Symposium*, pages 375–388. Internet Society, 2010.

[19] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pages 28–38, 1986.

[20] google-caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`.

[21] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In J. Palsberg, editor, *Proc. 37th ACM Symp. POPL*, pages 353–364, Madrid, Spain, Jan. 2010. ACM Press.

[22] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *13th ECOOP*, volume 1628 of *LNCS*, pages 205–229, Lisbon, Portugal, June 1999. Springer.

[23] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In D'Hondt [9].

[24] P. Heidegger and P. Thiemann. Contract-driven testing of JavaScript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172, Malaga, Spain, June 2010. Springer.

[25] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In D'Hondt [9].

[26] P. Heidegger and P. Thiemann. A heuristic approach for computing effects. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns*, TOOLS'11, Zurich, Switzerland, June 2011. Springer. to appear, draft available at: `http://proglang.informatik.uni-freiburg.de/jscontest/`.

[27] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 208–225, Fuji Susono, Japan, Apr. 2006. Springer.

[28] R. L. B. Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In S. Arora and G. T. Leavens, editors, *Proc. 24th ACM Conf. OOPSLA*, pages 97–116, Orlando, Florida, USA, 2009. ACM Press, New York.

[29] R. Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Santa Barbara, CA, USA, 1998.

[30] P. Le Hégaret, R. Whitmer, and L. Wood. W3C document object model. `http://www.w3.org/DOM/`, Aug. 2003.

[31] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[32] H. Lehner and P. Müller. Efficient runtime assertion checking of assignable clauses with datagroups. In D. S. Rosenblum and G. Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 338–352. Springer, 2010.

[33] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In S. Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, pages 359–371, New York, NY, USA, Jan. 2006. ACM.

[34] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS'09: Proceedings of the 14th European Conference on Research in Computer Security*, pages 505–522, Saint-Malo, France, 2009. Springer-Verlag.

[35] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

[36] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[37] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[38] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.

[39] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP*, volume 1445 of *LNCS*, pages 158–185, Brussels, Belgium, July 1998. Springer.

[40] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[41] F. Spoto and E. Poll. Static Analysis of JML's assignable Clauses. *International Workshop on Foundations of Object-Oriented Languages*, Jan. 2003.

[42] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[43] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer-Verlag.

[44] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In B. Pierce, editor, *Proc. 36th ACM Symp. POPL*, pages 41–52, Savannah, GA, USA, Jan. 2009. ACM Press.

[45] T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java. In *Proc. 18th ACM Conf. OOPSLA*, pages 135–148, Anaheim, CA, USA, 2003. ACM Press, New York.

A. Location-Based Semantics

The idea of the location-based semantics is to assign read or write permissions to the properties of a set of locations that is determined by a particular access permission. To do that safely requires a partial traversal of the object graph for the installation of each access permission.

This section contains the outline of an implementation that requires two main data structures and that relies on the instrumentation of all operations that affect the heap: creation of new objects, reading a property, and writing a property. It simplifies matters by assuming that each permission is represented by a *finite set* of classified access paths. To fully deal with the path notation from Sec. 1 and 2 introduces further problems in connection with cyclic object graphs. In particular, it might be necessary to implement the $*$ operator in permissions using backtracking.

Both of the two data structures are stacks. They have one entry for each currently installed access permission. The top entry contains the information for the most recently installed permission.

```
newlocs     : Stack of (Set of Location)
```

The top entry contains the newly allocated locations since the last active permission was installed. All stack entries are disjoint.

The rationale for this data structure is that newly allocated locations are not governed by previously installed permissions. Therefore, the program is free to read and write to their properties.

```
type Permission = {None, Readable, Writable}
     ordered by None < Readable < Writable
permissions : Stack of
     (Map from (Location x Property) to Permission)
```

The top entry contains the currently active permissions. The map on top of the stack is always less permissive than the map below on the intersection of the domains of the two maps. The map on top never contains locations in top(newlocs). The default value in a map is None.

These definitions are sufficient to define the instrumentation of the code. The notation store[l][p] addresses the object store at location l and property p.

```
// replacement for read operation
function read (l : Location, p : Property) {
  if (getPermission (l, p) >= Readable) {
    return store[l][p];
  } else {
    throw ReadViolation;
  }
}
// replacement for write operation
function write (l : Location, p : Property, x : Any) {
  if (getPermission (l, p) >= Writable) {
    store[l][p] = x;
  } else {
    throw WriteViolation;
  }
}
// replacement for new
function new() {
  l = some location not in dom(store);
  store[l] = empty object;
  top (newlocs) = top (newlocs) + {l};
}
```

Computing the access permission amounts to first checking if the location is new, in which case reading and writing is permitted, and then look it up in the permissions stack.

```
function getPermission (l : Location, p : Property) {
  if (l in top (newlocs)) {
    return Writable;
  } else {
    return top (permissions) (l, p);
  }
}
```

The remaining procedures implement the installation of a permission contract as shown in Sec. 1 and 2. It contains the extensions for path exceptions as mentioned in Sec. 6.

For simplicity, a permission is expressed as a base object location and a set of paths, where the internal steps of a path imply read permission and the final step implies write permission.

```
function installPermission (
    permissionsToGrant : List of (Location x (Set of Path)),
    permissionsToRevoke : List of (Location x (Set of Path))) {
  newPerm = new EmptyPermission();
  for ((l, P) in permissionsToGrant) {
    permitPaths (newPerm, l, P);
  }
  for ((l, P) in permissionsToRevoke) {
    forbidPaths (newPerm, l, P);
  }
  push (permissions, newPerm);
}
```

Uninstalling a permission (i.e., leaving its extent) happens in two steps. First, the most recently installed permissions/restrictions are withdrawn. Second, the newly allocated locations are joined.

```
function uninstallPermission () {
  pop (permission);
  newLocationsSinceInstallation = top (newlocs);
  pop (newlocs);
  top (newlocs) =
    union (top (newlocs), newLocationsSinceInstallation);
}
```

Defining new read and write permissions refines the existing permissions. Accesses that were forbidden before cannot be allowed by installing a new permission. Essentially, we traverse the object graph starting from the base object. This pseudocode does not address cycles in object graphs, which presents further challenges.

```
// install a set of permissions P on location l
function permitPaths (newPerm, l, P) {
  for (path in P) {
    permitSinglePath (newPerm, l, path);
  }
}
// pattern matching on path
function permitSinglePath (newPerm, l, path) {
  if (path is empty) {
    return;
  } else if (path is single property p) {
    // grant write permission if property was Writable before
    newPerm (l, p) = join (newPerm (l, p), getPermission (l, p));
  } else {
    path has the form p . path';
    if (getPermission (l, p) != None) {
      newPerm (l, p) = join (newPerm (l, p), Readable);
      permitSinglePath (newPerm, store[l][p], path');
    }
  }
}
```

Installing new exceptions is analogous to installing permissions. It must be applied *after* granting permissions as shown in installPermission. When processing an exception, only the last step of a path is forbidden.

```
// remove a set of permissions P on location l
function forbidPaths (newPerm, l, P) {
  for (path in P) {
    forbidSinglePath (newPerm, l, path);
  }
}
// pattern matching on path
function forbidSinglePath (newPerm, l, path) {
  if (path is empty) {
    return;
  } else if (path is single property p) {
    newPerm (l, p) = None;
  } else {
    path has the form p . path';
    // ignore existing permission for (l,p) because
    // there may be a different way to get to the
    // end of this path (via some alias)
    forbidSinglePath (newPerm, store[l][p], path');
  }
}
```

B. Further Examples

B.1 Introduction

An access contract explicitly states the set of paths (sequences of property accesses) that a method may access from the objects in scope. Being able to state such contracts is important in a language like JavaScript, where a side effect is the raison d'être of many operations. To support this claim, consider the following code:

```
function redirectTo (url) {
  window.location = url;
}
```

The type-signature contract (string) → **undefined** would be suitable for redirect, stating that the argument must be a string and that the undefined result value should be returned.[10] However, the interesting information about the function is that it changes the location property of the window object, which has the further effect of redirecting the web browser to a new page. To specify this effect, our extended contract language enables us to extend the above contract with an *access permission*:

```
... with [window.location]
```

This extended contract allows the function to access and modify the location property of window but denies access to any other object. Contract monitoring for such a contract enforces the permission at run time. For example, if the function's implementation above were replaced by

```
function redirectTo (url) {
  window.location = url;
  myhistory.push (url);
}
```

while keeping the same type signature and access permission, then monitoring would report a contract violation as soon as the function accesses the data structure myhistory.

B.2 Modular Layout Computation

Suppose you are a JavaScript developer who has just been assigned a maintenance task on a large AJAX application. In particular, you need to work on the code that performs a layout computation for a bunch of view objects. To start with, it would be advantageous to know which properties are modified by the code. Using our framework, a developer can gradually specify access contracts for the code until it runs without contract violation on a sufficiently large number of test cases. For example, the final specification may be as follows:

```
/*c {}.(int, int) → boolean with [this.x, this.y, this.w, this.h] */
Frame.prototype.layout = function (width, height) { ... }
```

The special comment /∗c ... ∗/ specifies a contract for a method. The part before **with** defines the type signature. In the subsequent access permission, this refers to the receiver object of the method call. The access paths specify that only properties named x,y, w, or h of the receiver object may be written.

An access path starts with any variable name in scope followed by a sequence of property names. It permits reading any property reachable by dereferencing some prefix of the access path and writing the properties reachable by dereferencing the entire access path. The special variable names this, $1, $2, ... refer to the receiver object of a method call and to the first, second, and so on parameter. They are synonymous to the respective parameter name.

B.3 Read-only Objects

Many libraries rely on a programming pattern to define JavaScript functions with keyword parameters. The idea is to define a function with one parameter which is always an object. The properties of this object play the role of keyword parameters as in this example:

```
c = createCanvas({width: 100, height: 200, background: "green"});
```

As it is generally considered bad programming style to assign to parameters, this parameter object should not be changed, either. Such changes could be forbidden with a contract:

```
/*c ({}) → undefined with [$1.*.@] */
```

This specification uses two new features in the access permission: as in name patterns for file access in a shell, the ∗ stands for any sequence of property names. The final @ stands for the empty set of property names. Thus, the first parameter must be read-only. Read permission is granted for all properties reachable from $1, but write permission is granted only for those access paths that end in a property name that is contained in the empty set, that is, for *no* access path.

B.4 Observer

In an implementation of the observer pattern, the programmer would like to make sure that an observer only reads and writes properties below the state component of the subject. This restriction may be expressed with the contract

```
/*c ({}) → any with [$1.state.*.?] */
Observer.prototype.update = function (subject) {
  ... subject.state.value = ...
}
```

With this access permission, any property below state is readable and writable but state itself is read-only. The final ? stands for any property name.

B.5 Regular Expression Permissions

Let's return to the example from the introduction, where we wanted to ensure that a method only accesses and modifies the window.location property. In the context of enforcement of security properties, it is more likely that we want to forbid access to a few chosen properties, whereas we do not care about accesses to the majority of properties. In such a situation, we might write an access permission like the following:

[10] undefined is a special value in JavaScript. Methods without an explicit return statement return undefined.

... **with** [window./^((?!status$).)/]

It specifies an access path that accepts read and write accesses only to properties of the window object that match the regular expression enclosed in slashes. The particular regular expression in the example matches all property names different from status. Often, such cases are easier to express with our syntax for revoking permissions, as in

... **with** [window.?] **except** [window.status,window.location]

which forbids accessing the status and location properties. Revocation is only possible for permissions granted in the same contract.

C. Syntax and Semantics of Effects

The implementation requires access permissions to be specified using the path notation informally introduced in Sec. B. We first present a formalization of this notation, which we then connect to the actual syntax used in the implementation.

Figure 10 presents the formal syntax of access permissions. An access permission classifies access paths π, which are sequences of property names. Access paths are classified as read paths, write paths, or negative paths by writing $\mathbf{R}(\pi)$, $\mathbf{W}(\pi)$, or $\mathbf{N}(\pi)$. An access permission is built from path permissions with set union and difference operators. A path permission b is either empty, a path step P followed by a permission, or an iterated path step P followed by a permission. P can be an arbitrary set of property names.

Figure 11 defines the semantics of access permissions with inference rules for the judgment $\kappa \prec a$, which indicates that the classified path κ matches permission a. Essentially, a single path step P in a permission is matched by a corresponding property $p \in P$ in the path. An iterated path step $P*$ is matched by a sequence of properties from P in the path. The three axioms on top of Fig. 11 implement the different treatment of the three kinds of path. A write path must be matched exactly by the permission, a read path may match any prefix of the permission, and a negative path only requires that a path prefix is matched by the permission. The latter choice is required for the implementation of the difference operator $a_1 - a_2$, where the second premise asks for $\mathbf{N}(\pi) \not\prec a_2$, that is, there should be no derivation of $\mathbf{N}(\pi) \prec a_2$. This definition enforces that the read language is prefix closed as well as the connection between the write and read languages mentioned in Sec. 3.1.

Turning to the concrete syntax, an access permission for a variable x has the following general form:

$$\textbf{with}\ [\mathsf{x.w}_1,\ldots,\mathsf{x.w}_n]\ \textbf{except}\ [\mathsf{x.e}_1,\ldots,\mathsf{x.e}_m] \qquad (3)$$

Translated to the formal syntax defined in Figure 10, this permission reads as follows:

$$a = (w_1 + \ldots + w_n) - e_1 - \ldots - e_m\ .$$

The access permission (3) corresponds to the language $\mathsf{L}_r = \{\pi \mid \mathbf{R}(\pi) \prec a\}$ of permitted read paths and the language $\mathsf{L}_w = \{\pi \mid \mathbf{W}(\pi) \prec a\}$ of permitted write paths for the variable x. Hence, adding a contract with an effect annotation to a function as in (3) is equivalent to surrounding the function body e with the permit expression **permit** x : L_r, L_w **in** e.

As in the formal syntax, paths of arbitrary length can be specified using the $*$ operator. For example, an access permission for x, x.next, x.next.next, ... for the elements of a list is written as x.next$*$. The wildcard property ? stands for the set of all property names. If the operator $*$ is used without a preceding property name, then it stands for ?$*$, specifying a sequence of arbitrary property names.

A property set can be specified in several ways. An identifier (as in x.test) or a string literal (x."foo.bar") specify singleton sets, with the string notation allowing special characters (like .) in property names. A regular expression (x./left|right/) specifies the set of properties that match the expression.

The implementation supports two further extensions. Regular expressions may also be used on the access path level (/x(.(left|right))$*$.data/). Further, it is possible to register a JavaScript callback to describe the path language in terms of JavaScript code. For example, the function f is called to test membership of a path in the language if the permission is js:f.

$$\begin{array}{llll}
p & \in & \mathit{Prop} & \text{property names} \\
\pi & ::= & \varepsilon \mid p.\pi & \text{access paths} \\
\gamma & ::= & \mathbf{R} \mid \mathbf{W} \mid \mathbf{N} & \text{access classifiers} \\
\kappa & ::= & \gamma(\pi) & \text{classified access path} \\
P & \subseteq & \mathit{Prop} & \text{set of property names} \\
b & ::= & \varepsilon \mid P.b \mid P*.b & \text{path permissions} \\
a & ::= & \emptyset \mid b \mid a + a \mid a - a & \text{access permissions} \\
\multicolumn{4}{l}{? = \mathit{Prop}, \qquad @ = \emptyset \subseteq \mathit{Prop}, \qquad *.b = ?*.b}
\end{array}$$

Figure 10. Syntax of access paths and access permissions.

$$\mathbf{W}(\varepsilon) \prec \varepsilon \qquad \mathbf{R}(\varepsilon) \prec b \qquad \mathbf{N}(\pi) \prec \varepsilon \qquad \frac{\gamma(\pi) \prec b \qquad p \in P}{\gamma(p.\pi) \prec P.b}$$

$$\frac{\gamma(\pi) \prec b}{\gamma(\pi) \prec P*.b} \qquad \frac{\gamma(\pi) \prec P*.b \qquad p \in P}{\gamma(\pi.p) \prec P*.b}$$

$$\frac{\kappa \prec a_1}{\kappa \prec a_1 + a_2} \qquad \frac{\kappa \prec a_2}{\kappa \prec a_1 + a_2} \qquad \frac{\gamma(\pi) \prec a_1 \qquad \mathbf{N}(\pi) \not\prec a_2}{\gamma(\pi) \prec a_1 - a_2}$$

$$\frac{(\forall \kappa \in K)\ \kappa \prec a}{K \prec a}$$

Figure 11. Matching paths with access permissions.

D. Case Study: Deltablue Benchmark

For a final case study, we tested the Deltablue benchmark which is also taken from the V8 benchmark suite.[11] It implements a constraint-solving algorithm for a hierarchy of objects. As a particularity, the constraint model is built by side-effects from constructors. The code implements 59 functions in 670 LOC. A person without prior knowledge of the code under test provided the contracts and implemented custom generators in about 4 hours. Here, the major complication was in reengineering the object hierarchies. The access permissions were automatically inferred and added within seconds.

Table 3 shows the result of testing about 830 mutated versions. In contrast to case studies so far, it is not possible to run unit tests for single functions or methods as the application heavily relies on global variables for storing state. Further, the major computations are triggered in the constructors of the different constraints. To test the application, we therefore utilized the actual benchmark application which consists of several test cases. The table also contains the number of exceptions that were triggered by the applications due to failed invariants in the constraint solver's internal state.

The type of the main function is /$*$c undefined $\rightarrow$ undefined $*$/. Hence it did not lead to any type contract error.

[11] `http://v8.googlecode.com/svn/data/benchmarks/v6/deltablue.js`

	type		type + effect	
fulfilled contracts	176	21.3%	102	12.3%
rejected contracts	626	75.6%	697	84.2%
reason for rejection				
signaled error	505	61.0%	469	56.6%
browser timeout	26	3.2%	29	3.5%
app exception	121	14.6%	73	8.8%
read violation	0	0.0%	135	16.3%
write violation	0	0.0%	20	2.4%

Table 3. Testing random mutations of the Richards case study.

For this application, adding access permissions increased the detection rate from 75.6% to 84.2%, which amounts to a 11.37% improvement.

E. Pre-State Snapshot

Proof: (Theorem 3.1) By induction on the derivation.

The only rule requiring non-trivial reasoning is GET. By induction, we can assume that H' and $(l, \mathcal{M})$ contain correct path information. We now have to show that the override operation creates a new correct value. Without loss of generalization, we can assume that the heap contains a reference, since otherwise the override operation is trivial. Hence, it holds that $H'(l)(p) = (u, (\ell', \mathcal{N}))$, and the result of the heap lookup is

$$(\ell', \mathcal{M}.p \otimes_u \mathcal{N})$$

For a given arbitrary fixed time stamp u' there are now two cases:

- $u' \in \mathit{dom}(\mathcal{N})$: In this case, the override operation picks the path from $\mathcal{N}$. This path is valid by induction.
- $u' \notin \mathit{dom}(\mathcal{N})$: If $u < u'$, the conclusion is trivial as $\mathcal{M}$ was path consistent. We do not need to consider the case $u \geq u'$, because in this case the map would not be defined and the precondition of the theorem would not hold.

□

Theorem 3.1 states the correctness of the path information, but does not yield the completeness of the gathered path information.

However, it is easy to see that the system does not drop any path information. The only rule that removes paths from the system is the third case of the override operation. Due to the condition for the third case (the time stamp u is not smaller that u') we can conclude that this case only arises if the property was written after the installation of the permit operation corresponding to the timestamp u. This write operation has stored a value inside the heap which was coupled with valid path information. If the value was reachable with respect to the heap with time stamp u, this path information is stored in the heap (and the first case of the override operation would trigger). The value stored in the heap by the write operation was not reachable in the heap with time stamp u. Thus, it is safe to remove the path from the map. For a more formal approach to completeness see Sec. G.

F. Stability of Violation

To prove Theorem 3.2, we first formulate a helping theorem.

Theorem F.1 *If $H_1 \succcurlyeq_\gamma H_2$ and $\rho_1 \succcurlyeq_\gamma \rho_2$ and*

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1' \tag{4}$$

then either

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \hookrightarrow H_2'; u_2'; v_2' \tag{5}$$

such that there exists γ' extending γ where $H_1 \succcurlyeq_{\gamma'} H_2$ and $u_1' = u_2'$ and $v_1' \succcurlyeq_{\gamma'} v_2'$ or

$$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e \Uparrow^O \tag{6}$$

such that the derivation of (6) *ends in an inconsistent read operation with respect to* (4)*.*

Proof: (Theorem F.1) By induction on the derivation of $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \hookrightarrow H_1'; u_1'; v_1'$.

Case VAR, $e \equiv x$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; x \hookrightarrow H_1; u; \rho_1(x)$
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; x \hookrightarrow H_2; u; \rho_2(x)$
Since $\rho_1 \succcurlyeq_\gamma \rho_2$, it holds that $\rho_1(x) \succcurlyeq_\gamma \rho_2(x)$.

Case LAM, $e \equiv \lambda x.e'$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; \lambda x.e' \hookrightarrow H_1; u; (\rho_1 \downarrow_{FV(\lambda x.e')}, \lambda x.e')$
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; \lambda x.e' \hookrightarrow H_2; u; (\rho_2 \downarrow_{FV(\lambda x.e')}, \lambda x.e')$
Since $\rho_1 \succcurlyeq_\gamma \rho_2$, it holds that $\rho_1 \downarrow_X \succcurlyeq_\gamma \rho_2 \downarrow_X$, for any set X of variables.

Case APP, $e \equiv e_0(e_1)$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_0(e_1) \hookrightarrow H_1'''; u_1'''; v_1$ because

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_0 \hookrightarrow H_1'; u'; (\rho_1', \lambda x.e') \tag{7}$$

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1'; u'; e_1 \hookrightarrow H_1''; u''; v_1' \tag{8}$$

$$\rho_1'[x \mapsto v_1'], \mathcal{R}, \mathcal{W} \vdash H_1''; u''; e' \hookrightarrow H_1'''; u'''; v_1 \tag{9}$$

By induction on (7), we obtain that either e_0 crashes on H_2 (which would make the whole application crash) or
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_0 \hookrightarrow H_2'; u'; (\rho_2', \lambda x.e')$
where, for some extension γ' of γ, $H_1' \succcurlyeq_{\gamma'} H_2'$ and $\rho_1' \succcurlyeq_{\gamma'} \rho_2'$.

By induction on (8), we obtain that either e_1 crashes on H_2' (which would make the whole application crash) or
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2'; u'; e_1 \hookrightarrow H_2''; u''; v_2'$
where, for some extension γ'' of γ', $H_1'' \succcurlyeq_{\gamma''} H_2''$ and $v_1' \succcurlyeq_{\gamma''} v_2'$.

By induction on (9), we obtain that either e' crashes on H_2'' (which would make the whole application crash) or
$\rho_2'[x \mapsto v_2'], \mathcal{R}, \mathcal{W} \vdash H_2''; u''; e' \hookrightarrow H_2'''; u'''; v_2$
where, for some extension γ''' of γ'', $H_1''' \succcurlyeq_{\gamma'''} H_2'''$ and $v_1 \succcurlyeq_{\gamma'''} v_2$.

Hence, rule APP yields
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_0(e_1) \hookrightarrow H_2'''; u'''; v_2$
where, for some extension γ''' of γ, $H_1''' \succcurlyeq_{\gamma'''} H_2'''$ and $v_1 \succcurlyeq_{\gamma'''} v_2$.

Case NEW, $e \equiv \mathtt{new}$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; \mathtt{new} \hookrightarrow H_1[\ell_1 \mapsto \emptyset]; u; (\ell_1, \emptyset)$ where $\ell_1 \notin \mathit{dom}(H_1)$

Clearly, there exists $\ell_2 \notin \mathit{dom}(H_2)$ so that NEW is applicable to H_2 yielding
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; \mathtt{new} \hookrightarrow H_2[\ell_2 \mapsto \emptyset]; u; (\ell_2, \emptyset)$

As $\gamma' = \gamma[\ell_1 \mapsto \ell_2]$ is an extension of γ it follows that $H_1[\ell_1 \mapsto \emptyset] \succcurlyeq_{\gamma'} H_2[\ell_2 \mapsto \emptyset]$ and $(\ell_1, \emptyset) \succcurlyeq_{\gamma'} (\ell_2, \emptyset)$.

Case GET, $e \equiv e'.p$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e'.p \hookrightarrow H_1'; u'; v_1'$
because

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e' \hookrightarrow H_1'; u'; (\ell_1, \mathcal{M}_1) \tag{10}$$

$$\mathcal{R} \vdash_{\text{chk}} \mathcal{M}_1.p \tag{11}$$

where $v_1' = \mathcal{M}_1.p \otimes H_1'(\ell_1)(p)$

Induction on (10) yields that either e' crashes on H_2 (making the whole evaluation crash via GET-CRASH1) or

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}\ \textsc{app-crash1}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}\ \textsc{app-crash2}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \hookrightarrow H''; u''; v_1 \qquad \rho'[x \mapsto v_1], \mathcal{R}, \mathcal{W} \vdash H''; u''; e \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \Uparrow^i}\ \textsc{app-crash3}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^i}\ \textsc{get-crash1}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \not\vdash_{\text{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^R}\ \textsc{get-crash2}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \mathcal{R} \vdash_{\text{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \Uparrow^O}\ \textsc{get-crash3}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^i}\ \textsc{put-crash1}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^i}\ \textsc{put-crash2}$$

$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \qquad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \qquad \mathcal{W} \not\vdash_{\text{chk}} \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \Uparrow^W}\ \textsc{put-crash3}$$

$$\frac{\rho[x \mapsto \rho(x) \lhd [u \mapsto \varepsilon]], \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H; u+1; e \Uparrow^i}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e \Uparrow^i}\ \textsc{permit-crash}$$

Figure 12. Crashing and partial computations.

$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e' \hookrightarrow H_2'; u'; (\ell_2, \mathcal{M}_2)$
where $H_1' \succcurlyeq_{\gamma'} H_2'$ and $(\ell_1, \mathcal{M}_1) \succcurlyeq_{\gamma'} (\ell_2, \mathcal{M}_2)$ for some γ' extending γ.

That means $\ell_2 = \gamma'(\ell_1)$ and $\mathcal{M}_1 = \mathcal{M}_2$. The latter implies with (11) that
$\mathcal{R} \vdash_{\text{chk}} \mathcal{M}_2.p$
and it remains to consider $v_2' = \mathcal{M}_2.p \otimes H_2'(\ell_2)(p)$.

Let $(u_1, v_1) = H_1'(\ell_1)(p)$ and $(u_2, v_2) = H_2'(\ell_2)(p)$.

If $u_1 = u_2$, then $H_1' \succcurlyeq_{\gamma'} H_2'$ implies $v_1 \succcurlyeq_{\gamma'} v_2$ which further implies $v_1' = \mathcal{M}_1.p \otimes (u_1, v_1) \succcurlyeq_{\gamma'} v_2' = \mathcal{M}_2.p \otimes (u_2, v_2)$.

If $u_1 < u_2$, then this read operation is inconsistent with respect to $H_1 \succcurlyeq_\gamma H_2$ and we choose the non-deterministic error exit by continuing the derivation with rule GET-CRASH3.

Case PUT, $e \equiv$:
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_1'.p := e_2' \hookrightarrow H_1'''; u'''; v_1'$ because

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_1' \hookrightarrow H_1'; u'; (\ell_1, \mathcal{M}_1) \tag{12}$$
$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1'; u'; e_2' \hookrightarrow H_1''; u''; v_1' \tag{13}$$
$$\mathcal{W} \vdash_{\text{chk}} \mathcal{M}_1.p \tag{14}$$
$$H_1''' = H_1''[\ell_1 \mapsto H_1''(\ell_2)[p \mapsto (u'', v_1')]] \tag{15}$$
$$u''' = u'' + 1 \tag{16}$$

By induction on (12), we have that either $\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_1' \Uparrow^i$ or
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_1' \hookrightarrow H_2'; u', (\ell_2, \mathcal{M}_2)$
such that $H_1' \succcurlyeq_{\gamma'} H_2'$ and $\ell_2 = \gamma'(\ell_1)$, for some extension γ' of γ.

In the latter case, we continue by induction on (13). We have that either $\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2'; u'; e_2' \Uparrow$ or $\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2'; u'; e_2' \hookrightarrow H_2''; u''; v_2'$
such that $H_1'' \succcurlyeq_{\gamma'} H_2''$ and $\ell_2 = \gamma'(\ell_1)$, for some extension γ' of γ.

In the latter case, $\mathcal{W} \vdash_{\text{chk}} \mathcal{M}_2.p$ because $\mathcal{M}_1 = \mathcal{M}_2$ and it remains to show that

$$H_1''[\ell_1 \mapsto H_1''(\ell_1)[p \mapsto (u'', v_1')]]$$
$$\succcurlyeq_{\gamma'} H_2''[\ell_2 \mapsto H_2''(\ell_2)[p \mapsto (u'', v_2')]]$$

which is clear from the definition: we are overwriting one property in one object in a way that the time stamps are identical and with related values.

Hence, rule PUT is applicable to complete the derivation.

Case PERMIT, $e \equiv \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e'$:

Given that $\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e' \hookrightarrow H_1'; u'; v_1'$ it must be that

$$\rho_1', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H_1; u+1; e' \hookrightarrow H_1'; u'; v_1 \tag{17}$$
$$\rho_1' = \rho_1[x \mapsto \rho_1(x) \lhd [u \mapsto \varepsilon]] \tag{18}$$

By induction on (17), it must be that either
$\rho_2', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H_2; u+1; e' \Uparrow^i$, in which case the whole expression crashes by rule PERMIT-CRASH, or
$\rho_2', \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H_2; u+1; e' \hookrightarrow H_2'; u'; v_2$ where $H_1' \succcurlyeq_{\gamma'} H_2'$ and $v_1 \succcurlyeq_{\gamma'} v_2$ for some γ' extending γ. □

Proof: (Theorem 3.2) By induction on the derivation of
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e \Uparrow^i$

The proof relies on Theorem F.1 to handle all non-crashing subcomputations. Hence, the induction only handles the crashing rules in Fig. 12.

It is interesting to observe that the $\Uparrow^O$ outcome only arises due to subcomputations that did not crash in H_1. So they are only generated by invocations of Theorem F.1, not by cases handled directly in this proof.

Case APP-CRASH1, $e \equiv e_0(e_1)$:
Immediate by appeal to the induction hypothesis.

Case APP-CRASH2, $e \equiv e_0(e_1)$:
By inversion of rule APP-CRASH2, we obtain

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e_0 \hookrightarrow H_1'; u'; (\rho_1', \lambda x.e') \quad (19)$$

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1'; u'; e_1 \Uparrow^i \quad (20)$$

By Theorem F.1 applied to (19), we obtain either
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_0 \Uparrow^O$, in which case we complete the derivation with rule APP-CRASH1, or
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e_0 \hookrightarrow H_2'; u'; (\rho_2', \lambda x.e')$
where $H_1' \succcurlyeq_{\gamma'} H_2'$ and $\rho_1' \succcurlyeq_{\gamma'} \rho_2'$ for some γ' extending γ.
Thus, we can apply induction to (20) to obtain
$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1'; u'; e_1 \Uparrow^j$
with the stated relation between i and j. Applying APP-CRASH2 completes the derivation.

Case APP-CRASH3, $e \equiv e_0(e_1)$:
Analogous to case APP-CRASH2.

Case GET-CRASH1, $e \equiv e'.p$:
Analogous to case APP-CRASH1.

Case GET-CRASH2, $e \equiv e'.p$:
By inversion, we obtain

$$\rho_1, \mathcal{R}, \mathcal{W} \vdash H_1; u; e' \hookrightarrow H_1'; u'; (\ell_1, \mathcal{M}_1) \quad (21)$$

$$\mathcal{R} \not\vdash_{\mathsf{chk}} \mathcal{M}_1.p \quad (22)$$

As in previous cases, either
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e' \Uparrow^j$ (which gets propagated) or
$\rho_2, \mathcal{R}, \mathcal{W} \vdash H_2; u; e' \hookrightarrow H_2'; u'; (\ell_2, \mathcal{M}_2)$
where $H_1' \succcurlyeq_{\gamma'} H_2'$ and $\rho_1' \succcurlyeq_{\gamma'} \rho_2'$ and $\ell_2 = \gamma'(\ell_1)$ and $\mathcal{M}_1 = \mathcal{M}_2$ for some γ' extending γ.
Hence, $\mathcal{R} \not\vdash_{\mathsf{chk}} \mathcal{M}_2.p$ and an application of GET-CRASH2 concludes the derivation.

Case GET-CRASH3: is not applicable.

Cases PUT-CRASH1, PUT-CRASH2, PUT-CRASH3, PERMIT-CRASH: Analogous to previous cases. □

G. Tracing Soundness

This section does not have a corresponding part in the paper. But the facts we prove here are interesting on its own.

The result underlines that our semantics adheres to the pre-state snapshot principle (Sec. 2.3). Informally, suppose an access contract is attached to a variable holding a reference ℓ to some object. Then we want to make sure that if an object at ℓ' is accessed via this variable without triggering a violation, then there is a path sanctioned by the contract from ℓ to ℓ' in the pre-state of the contract installation.

To formulate a precise statement, we extend the evaluation judgment to trace all read and write accesses in sets $T^r, T^w \subseteq Loc \times Prop$:

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; v\ [T^r, T^w]$$

Figure 13 shows the modified rules for property read and write; the remaining rules just union the trace sets from the subcomputations as shown in the PUT' rule.

We further need to refer to all heap locations reachable from a given object location. This notion is formalized with a mapping $\mathit{reach} : \mathit{Heap} \times \mathit{Val} \rightharpoonup \wp(\mathit{Loc})$, which returns the set of locations that are reachable from an input value v by dereferencing along any path $\pi \in \mathit{Path}$, using the auxiliary function $\Downarrow$ (see Figure 14). This function is heavily overloaded, but it just distributes the work. The first case accepts a set of paths and unions the results of each path. The second case accepts the result of a property read, a pair of a time stamp u and a value v, and returns the result for the value. The third case returns $\emptyset$ if the value is not a reference. Otherwise, it leaves the actual dereferencing to function $\Downarrow'$. The function $\Downarrow'$ is driven by its path argument. If the path is empty, it returns the object reached. Otherwise, it dereferences the first step in the path continuing with $\Downarrow$, case 2.

The acc function yields pairs of locations and property names for all accessible properties along a path $\pi \in \Pi$. These pairs express "last steps" (ℓ', p) in an access path $\pi.p$: ℓ' is the object reachable by path π and p is the property to be accessed. The first equation extends the function to sets of paths by joining the results on individual paths. The second equation deals with a reference value by dereferencing all steps of a path except the last one and pairing the resulting location with the last step of the path. The third equation handles a non-reference value.

The following theorem states the essence of the pre-state snapshot principle. To avoid excessive formal machinery, the statement is formulated in a setting where the variable to which the contract is attached refers to a part of the heap that is not reachable from other parts of the heap.

GET'

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T^r, T^w] \\ \mathcal{R} \vdash_{\mathsf{chk}} \mathcal{M}.p \qquad v' = \mathcal{M}.p \otimes H'(\ell)(p)\end{array}}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow' H'; u'; v'\ [T^r \cup \{(\ell, p)\}, T^w]}$$

PUT'

$$\frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T_1^r, T_1^w] \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow' H''; u''; v\ [T_2^r, T_2^w] \\ \mathcal{W} \vdash_{\mathsf{chk}} \mathcal{M}.p \qquad H''' = H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]\end{array}}{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \\ \hookrightarrow' H'''; u'' + 1; v\ [T_1^r \cup T_2^r, T_1^w \cup T_2^w \cup \{(\ell, p)\}]\end{array}}$$

Figure 13. Tracing property read and write.

$$\begin{array}{lcl}
\mathit{reach}(H, \{v_1, \dots, v_n\}) & = & \bigcup_i \mathit{reach}(H, v_i) \\
\mathit{reach}(H, v) & = & \Downarrow(H, v, \mathit{Path}) \\
\mathit{acc}(H, v, \Pi) & = & \bigcup\{\mathit{acc}(H, v, \pi) \mid \pi \in \Pi\} \\
\mathit{acc}(H, (\ell, \mathcal{M}), \pi.p) & = & \{(\ell', p) \mid \ell' \in \Downarrow'(H, \ell, \pi)\} \\
\mathit{acc}(H, v, \pi) & = & \emptyset \qquad \text{if } v \notin \mathit{Ref}
\end{array}$$

$$\begin{array}{lcl}
\Downarrow(H, v, \Pi) & = & \bigcup\{\Downarrow(H, v, \pi) \mid \pi \in \Pi\} \\
\Downarrow(H, (u, v), \pi) & = & \Downarrow(H, v, \pi) \\
\Downarrow(H, v, \pi) & = & \begin{cases} \Downarrow'(H, \ell, \pi) & v = (\ell, \mathcal{M}) \\ \emptyset & v \notin \mathit{Ref} \end{cases} \\
\Downarrow'(H, \ell, \varepsilon) & = & \{\ell\} \\
\Downarrow'(H, \ell, p.\pi) & = & \begin{cases} \Downarrow(H, H(\ell)(p), \pi) & p \in \mathit{dom}(H(\ell)) \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

Figure 14. Heap traversal.

Theorem G.1 *Suppose that* $\rho, \mathcal{R}, \mathcal{W} \vdash H_0; \mathtt{permit}\, x : L_r, L_w\, \mathtt{in}\, e \hookrightarrow' H_1; v\ [T^r, T^w]$ *and that* $\mathit{reach}(H_0, \rho(\mathit{FV}(e) \setminus \{x\})) \cap X = \emptyset$ *where* $X = \mathit{reach}(H_0, \rho(x))$.

Then $T^r \cap (X \times \mathit{Prop}) \subseteq \mathit{acc}(H_0, \rho(x), L_r)$ *and* $T^w \cap (X \times \mathit{Prop}) \subseteq \mathit{acc}(H_0, \rho(x), L_w)$.

The second assumption just says that x does not share with the remaining variables. The conclusion of the theorem says that for every access pair $(\ell, p) \in T_r$ where ℓ happens to be reachable from $\rho(x)$ this access must be sanctioned by the language L_r of read permissions. The latter is formalized via the *acc* function: it splits every access path in L_r in a prefix π and last property p, computes the dereferenced locations from $\rho(x)$ along path π, and pairs the results (at most one) with p.

The theorem clearly implies that accesses or modifications to newly allocated objects are not checked by the access contract.

To prove this theorem, we establish an invariant, which we formulate for the judgment without the traces because they are not needed to prove it. The assumption $\rho, \mathcal{R}, \mathcal{W} \vdash H_0; u_x; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e \hookrightarrow H_1; u_1; v$ in the theorem can only hold (by inversion) if its premise also holds:

$$\rho', \mathcal{R}[u_x \mapsto L_r], \mathcal{W}[u_x \mapsto L_w] \vdash H_0; u_x + 1; e \hookrightarrow H_1; u_1; v \tag{23}$$

where $\rho' = \rho[x \mapsto \rho(x) \lhd [u_x \mapsto \varepsilon]]$. Let's further assume that $\rho(x) = (\ell_x, m_x) \in \mathit{Ref}$ — otherwise, the theorem is trivially true because $v \notin \mathit{Ref} \Rightarrow \Downarrow(H_0, v, \pi) = \emptyset$, for all π, so that $X = \emptyset$.

Definition G.1 *A value v is* primarily reachable *(short: p.r.) from ℓ_x with index u_x in H_0 if either*

- $v = (\ell, \mathcal{M})$ *with $u_x \in \mathit{dom}(\mathcal{M})$ implies that* $\ell \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x))$,
- $v = (\rho, \lambda y.e')$ *with ρ primarily reachable, or*
- $v \in \mathit{Int}$.

An environment ρ is p.r. if $(\forall y \in \mathit{dom}(\rho))$ $\rho(y)$ is p.r. A heap H is p.r. if $\forall \ell \in \mathit{dom}(H)$ and $\forall p \in \mathit{dom}(H(\ell))$ $H(\ell)(p)$ p.r. (All with respect to the same fixed ℓ_x, u_x, and H_0.)

Lemma G.1 *For each judgment $\rho', \mathcal{R}', \mathcal{W}' \vdash H'; u'; e' \hookrightarrow H''; u''; v''$ occurring in the derivation of* (23) *it holds that: if ρ' and H' are p.r. from ℓ_x with index u_x in H_0, then so are H'' and v''.*

Proof: By induction on the derivation. Each case refers to the variables used in the respective rule in Figure 3.

Case VAR: obviously true.

Case LAM: obviously true.

Case APP: By the assumption on ρ and H, induction on e_0 yields H' and ρ' p.r. As now ρ and H' are p.r., induction yields that H'' and v_1 p.r. As $\rho'[x \mapsto v_1]$ and H'' are p.r., induction yields H''' and v p.r., which proves the result.

Case NEW: The heap $H[\ell \mapsto \emptyset]$ and the value $(\ell, \emptyset)$ are both p.r.

Case GET: By induction, H' and $(\ell, \mathcal{M})$ are p.r. But that means, if $u_x \in \mathit{dom}(\mathcal{M})$ then $\ell \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x))$. It remains to show that $\mathcal{M}.p \otimes H'(\ell)(p)$ is p.r. The only interesting case occurs if $H'(\ell)(p) = (u, (\ell', \mathcal{N}))$, in which case the returned value is $\mathcal{M}.p \otimes (u, (\ell', \mathcal{N})) = (\ell', \mathcal{M}.p \otimes_u \mathcal{N})$.

If $u_x \in \mathit{dom}(\mathcal{N})$, then the heap location has changed its content since the access permission associated with u_x and it has been overwritten with a value reachable in H_0 from ℓ_x on path $\mathcal{N}(u_x)$. This path annotation has to stay in force to ensure p.r. of the result: $(\mathcal{M}.p \otimes_u \mathcal{N})(u_x) = (\mathcal{N})(u_x)$, for which p.r. holds by the inductive assumption.

If $u_x \notin \mathit{dom}(\mathcal{N})$, then the contents of the heap location has not yet been reached from ℓ_x. There are two cases, which can be distinguished by comparing u and u_x. If $u \leq u_x$, then the heap location has not changed since H_0 and the result can be marked as visited. This is expressed by $(\mathcal{M}.p \otimes_u \mathcal{N})(u_x) = (\mathcal{M}.p)(u_x) = \mathcal{M}(u_x).p$. By the property read that happens in this rule, it is clear that $\ell' \in \Downarrow'(H_0, \ell_x, \mathcal{M}(u_x).p)$.

If, however, $u > u_x$, then the heap location has changed since H_0, but the new value has not been reachable from ℓ_x in H_0. For that reason, the value must not receive a u_x annotation. This is expressed by $(\mathcal{M}.p \otimes_u \mathcal{N})(u_x) =$undefined.

Case PUT: by induction H' and $(\ell, \mathcal{M})$ are p.r. Hence, H'' and v are also p.r. by induction. So is the final heap as the rule overwrites a value with a p.r. value.

Case PERMIT: immediate by induction. □

Towards the proof of Theorem G.1, which is by induction on the evaluation judgment with traces, we observe that the top-level judgment "seeds" the lemma in a non-trivial way. The environment $\rho[x \mapsto \rho(x) \lhd [u_x \mapsto \varepsilon]]$ is p.r. with respect to u_x, ℓ_x, and H_0 (from (23)) because $\ell_x \in \Downarrow'(H_0, \ell_x, \varepsilon)$ and no other environment entry refers to u_x. Similarly, the heap H_0 is p.r. because does not contain any reference to u_x. Thus, the lemma tells us that H_1 and v are also p.r.