Access Permission Contracts for Scripting Languages

Annette Bieniusa, Phillip Heidegger, Peter Thiemann

University of Freiburg, Germany

18 November 2011

Scripting Languages are Widely Used

Scripting languages provide

- simple access to powerful libraries
- end user programmability
  (simple concepts, dynamicity)
- quick combination of scripts
- quick development and evolution

Scripting Languages are Widely Used

Scripting languages provide

- simple access to powerful libraries
- end user programmability
  (simple concepts, dynamicity)
- quick combination of scripts
- quick development and evolution

Research Question

- What does it take to write and maintain reliable programs in a scripting language?
- What tools are useful?

Our Specimen: JavaScript

JavaScript is the language of the Web

Our Specimen: JavaScript

JavaScript is the language of the Web

91%

of all websites use JavaScript

(http://w3techs.com/ as of November 14, 2011)

Typical Scenario: Program Maintenance

- Programmer inherits reams of JavaScript code
- Task
  - Change / extend existing functionality
  - Implement a new feature
  - Fix a bug

Typical Scenario: Program Maintenance

- Programmer inherits reams of JavaScript code
- Task
  - Change / extend existing functionality
  - Implement a new feature
  - Fix a bug
- **Not** supported by a structured namespace . . .

Maintenance Questions

Program Understanding: Exploring Operations

- What is the public interface?
- What are the signatures?
- What changes are inflicted on the object graph?

Scenarios

Specification Levels for an Operation

1. The programmer provides the code
2. The programmer provides a type signature
3. The programmer provides a type signature with effects
4. The programmer provides a full specification

JSConTest:
Tool Support for **Partial Specifications**

Overview

- Type signatures and contracts for JavaScript with monitoring and random testing
- Effects for JavaScript: access permission contracts
- Effect inference

Type Signatures and Contracts

```
/*c int → int */
function f(x) { return 2 * x; };

/*c (int,int) → boolean */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

- contracts are checked / monitored at run time
- violations are flagged, e.g., if f or p is called on non-integer argument or if p does not return a boolean.

Object Types

An object type having at least the properties width, height, and background.

```
/*c ({width: int, height: int; background: string}) → undefined */
function createCanvas(arg) {
  ... arg.width * arg.height * screen.DEPTH ...
}
```

Object Types

An object type having at least the properties width, height, and background.

```
/*c ({width: int, height: int; background: string}) → undefined */
function createCanvas(arg) {
  ... arg.width * arg.height * screen.DEPTH ...
}
```

Method type where receiver type must have two integer properties, x and y.

```
/*c {x: int, y: int}.(int, int) → boolean */
Frame.prototype.layout = function (width, height) {
 ... this.x ... this.y ...
}
```

Random Testing from Type Contracts

- Observation (QuickCheck): Types are a good basis for generating random test data

Random Testing from Type Contracts

- Observation (QuickCheck): Types are a good basis for generating random test data
- Type contracts are just as good
- Contracts in negative positions serve as generators; contracts in positive positions serve as checkers

Example for Random Testing

```
/*c (int,int) → bool */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

For testing this function, the context needs to provide a pair of integers and needs to check the return value for a boolean.

Example for Random Testing

```
/*c (int,int) → bool */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

For testing this function, the context needs to provide a pair of integers and needs to check the return value for a boolean.

Pitfall

What is the probability that random testing finds the problem?

Pitfall for Random Testing

```
/*c (int,int) → bool */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

- random generator for int uniformly distributed

$\Rightarrow$ $P(x = 10) \approx 2^{-32}$

$\Rightarrow$ uniformly distributed generators are not always a good choice

Guided Random Testing

```
/*c (int@numbers,int) → bool */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

Annotate the int contract with @numbers.

Guided Random Testing

```
/*c (int@numbers,int) → bool */
function p(x,y) {
  if (x != y) {
    if (f(x) == x + 10) return "false"; // contract violation
  };
  return true;
};
```

Annotate the int contract with @numbers.

- ⇒ Changes the probability distribution
- ⇒ Generates random expressions with numbers from the source program
- ⇒ Usually locates the violation in less than 10 test runs
- ⇒ Highly effective also for more complicated conditions

Guided Contract for Objects

```
/*c (object@labels) → bool */
function h(x) {
  if (x && x.p && x.quest)
    return "false"; // contract violation
  return true;
};
```

- Random generation of objects; presence of particular labels unlikely

Guided Contract for Objects

```
/*c (object@labels) → bool */
function h(x) {
  if (x && x.p && x.quest)
    return "false"; // contract violation
  return true;
};
```

- Random generation of objects; presence of particular labels unlikely
- Annotation @labels
- Generator prefers to use the labels inside of the function body

⇒ Raises probability to generate a property with names p or quest; locates the violation

Effects: Access Permission Contracts

- Type contracts are not sufficiently expressive
- Effect of operation describes the locations read and written by it
- Expressed by **access permission contract**

Effects: Access Permission Contracts

- Type contracts are not sufficiently expressive
- Effect of operation describes the locations read and written by it
- Expressed by **access permission contract**

Access Permission

- Abstraction of a set of access paths
- Syntax: file path with wildcards, components are property names

Effects: Access Permission Contracts

- Type contracts are not sufficiently expressive
- Effect of operation describes the locations read and written by it
- Expressed by **access permission contract**

Access Permission

- Abstraction of a set of access paths
- Syntax: file path with wildcards, components are property names

... Contract

- comes with dynamic monitoring
- see `http://proglang.informatik.uni-freiburg.de/jscontest/`

Example: Add Node to Singly-Linked List

How does this operation affect the object graph?

```
function add(data) {
    var node = {data: data, next: null}, current;
    if (this._head === null) {
        this._head = node;
    } else {
        current = this._head;
        while(current.next) { current = current.next; }
        current.next = node;
    }
    this._length++;
}
```

Example: Add Node to Singly-Linked List

How does this operation affect the object graph?

```
function add(data) {
    var node = {data: data, next: null}, current;
    if (this._head === null) {
        this._head = node;
    } else {
        current = this._head;
        while(current.next) { current = current.next; }
        current.next = node;
    }
    this._length++;
}
```

- Reads and writes this._head and this._length.

Example: Add Node to Singly-Linked List

How does this operation affect the object graph?

```
function add(data) {
    var node = {data: data, next: null}, current;
    if (this._head === null) {
        this._head = node;
    } else {
        current = this._head;
        while(current.next) { current = current.next; }
        current.next = node;
    }
    this._length++;
}
```

- Reads and writes this._head and this._length.
- Reads this._head.next...next and writes the last next property

Example: Add Node to Singly-Linked List

How does this operation affect the object graph?

```
function add(data) {
    var node = {data: data, next: null}, current;
    if (this._head === null) {
        this._head = node;
    } else {
        current = this._head;
        while(current.next) { current = current.next; }
        current.next = node;
    }
    this._length++;
}
```

- Reads and writes this._head and this._length.
- Reads this._head.next...next and writes the last next property
- Does not access the data argument

Example with Access Permission Contract

```
/*c {}.(any) → undefined
    with [this._head, this._head.next*.next, this._length] */
function add(data) {
    var node = {data: data, next: null}, current;
    if (this._head === null) {
        this._head = node;
    } else {
        current = this._head;
        while(current.next) { current = current.next; }
        current.next = node;
    }
    this._length++;
}
```

Effects for Singly-Linked List Library

- add(data):
  this._head, this._head.next∗.next, this._length
- item(index):
  this._length.@, this._head.next∗.next.@, this._head.next∗.data.@
- remove(index):
  this._head.next∗.data.@, this._head.next∗.next, this._length
- size():
  this._length.@
- toArray():
  this._head.next∗.next.@
- toString():
  this._head.next∗.next.@

Syntax of Access Permissions

$$
\begin{array}{llll}
P & \subseteq & Prop, \quad p \in Prop & \text{properties} \\
 & & & \\
b & ::= & \varepsilon \mid P.b \mid P*.b & \text{path permissions} \\
a & ::= & \emptyset \mid b \mid a + a & \text{access permissions} \\
 & & & \\
\pi & ::= & \varepsilon \mid p.\pi & \text{access paths} \\
\gamma & ::= & \mathbf{R} \mid \mathbf{W} & \text{access classifiers} \\
\kappa & ::= & \gamma(\pi) & \text{classified access paths}
\end{array}
$$

$$? = Prop, \qquad @ = \emptyset \subseteq Prop$$

Path Semantics of Access Permissions

$\boxed{\gamma(\pi) \prec a}$ path $\gamma(\pi)$ matches permission a

$$\mathbf{W}(\varepsilon) \prec \varepsilon \qquad \mathbf{R}(\varepsilon) \prec b$$

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

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

$$\frac{\kappa \prec a_1}{\kappa \prec a_1 + a_2} \qquad \frac{\kappa \prec a_2}{\kappa \prec a_1 + a_2} \qquad \frac{(\forall \kappa \in K)\ \kappa \prec a}{K \prec a}$$

Examples

- $\mathbf{W}(\text{this.head}) \prec \text{this.head}$
- $\mathbf{W}(\text{this.length}) \not\prec \text{this.length.@}$ because $\mathbf{W}(\varepsilon) \not\prec @$
- $\mathbf{R}(\text{this.length}) \prec \text{this.length.@}$ because $\mathbf{R}(\varepsilon) \prec @$

Properties

1. If $\mathbf{R}(\pi.p) \prec a$, then $\mathbf{R}(\pi) \prec a$
   Read permissions are closed under prefix.
2. If $\mathbf{W}(\pi.p) \prec a$, then $\mathbf{R}(\pi) \prec a$
   The initial segment of a write permission yields read permission.
3. $\mathbf{W}(\pi) \not\prec b.@$
   A path permission ending in @ indicates read-only access.

All Settled?

- At this point, the issue seems settled.
- The semantics of an access path seems obvious and intuitive.
- Is it?

Interpretation of Paths

```
/∗c (obj, obj) → any with [x.b,y.a] ∗/
function h(x, y) {
  y.a = 1;
  y.b = 2; // violation?
}
// entry point #1
function h1() {
  var o = { a: −1, b: −2 };
  h(o, o);
}
// entry point #2
function h2() {
  h({ a: −1, b: −2 }, { a: −1, b: −2 });
}
```

Design Space for Semantics I

Path-dependent access

vs.

Location-dependent access

Path-Dependent Access

An access permission grants the right to read or modify a property of an object depending on the actual traversal path through which the object has been reached.

$\Rightarrow$ For each access, there is a unique path that determines the access rights.

$\Rightarrow$ `h1()` and `h2()` both lead to violation

Location-Dependent Access

An access permission attaches the right to read or modify a property of an object to its location.

⇒ For each access, there may be a number of paths in the permission that contribute to the access rights.

⇒ h1() is accepted because y.b is an alias of x.b, but h2() leads to a violation.

⇒ Violation is not stable

Design Space for Semantics, II

Dynamic Extent

vs.

Lexical Extent

Example I

```
/*c (obj) → any with [x.a] */
function d1(x) {
  return x.a; // violation if called from d2
}
/*c (obj) → any with [] */
function d2(x) {
  return d1(x);
}
```

Example I

```
/*c (obj) → any with [x.a] */
function d1(x) {
  return x.a; // violation if called from d2
}
/*c (obj) → any with [] */
function d2(x) {
  return d1(x);
}
```

- dynamic extent: the restriction imposed by contract on d2 carries over to d1
- lexical extent: ?

Example II

```
/*c (obj) → (() → any) with [x.b] */
function f(x) {
  return function() { return x.a + " " + x.b; };
}
function f1() {
  var r = f({ a: "secret", b: "revealed" });
  return r();
}
```

Example II

```
/*c (obj) → (() → any) with [x.b] */
function f(x) {
  return function() { return x.a + ” ” + x.b; };
}
function f1() {
  var r = f({ a: ”secret”, b: ”revealed” });
  return r();
}
```

- dynamic extent: no violation
- lexical extent: reading x.a triggers violation

Example II

```
/*c (obj) → (() → any) with [x.b] */
function f(x) {
  return function() { return x.a + ” ” + x.b; };
}
function f1() {
  var r = f({ a: ”secret”, b: ”revealed” });
  return r();
}
```

- dynamic extent: no violation
- lexical extent: reading x.a triggers violation
- If x.a should not be read, this contract is more appropriate

```
/*c (obj) → (() → any with [x.b]) with [x.b] */
```

Design Space for Semantics, III

Pre-State Snapshot

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

Alternatives to Pre-State Snapshot?

Candidates for Reference Heaps

- pre-state
  consistent with verification approaches (precondition)
- current state
  “symbolic” interpretation of access paths
- post-state
  ???

Symbolic Interpretation of Access Paths

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

Symbolic Interpretation of Access Paths

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

- Expectation: x.a.b does not change
- Admitted by symbolic interpretation: inconsistent with verification

Design Space for Semantics, IV

Sticky Update

A property assignment keeps the access paths of the value on its right-hand side.

Consequence of Sticky Update

```
/*c (obj) → any with [x.a,x.b.a] */
function l(x) {
  x.a = x.b;
  x.a.a = 42;
}
function l1() {
  var x = { a: {}, b: {}};
  l(x);
}
```

Is there a violation?

Design Choices in JSContest

Objective: Partial Specifications

- Path-dependent access
- Dynamic extent
- Pre-state snapshot
- Sticky update

Choices consistent with static analysis (effect systems) and static verification (c.f. dynamic frame rule by Smans et al)

Alternative Design Choices

Objective: Security

Different choices seem advantageous

- location-based semantics
- lexical extent (?)
- access **restrictions** instead of permissions
  i.e., guarantee no access to window.location

Side Remark: Efficiency

Path-Dependent Access

- references need to be paired with path information
- checking a permission: $O(\sharp\text{installed permissions})$
- installing a permission: $O(1)$

Location-Dependent Access

- separate data structure for rights management
- checking: $O(1)$
- installing a permission: (multiple) heap traversals; does that amortize?

Technical Development

Big-step evaluation judgment

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

- ρ environment
- $\mathcal{R}$, $\mathcal{W}$ read and write permissions
- H, H' heap
- u, u' timp stamp
- e expression
- v return value

Two rules

PERMIT

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

GET

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

Theorem: Soundness

- Reference value $::=$ Location $\times$ PMap
- PMap $::=$ Stamp $\rightarrow$ AccessPath

Theorem

For each reference value, the access path information is correct with respect to its corresponding pre-state heap.

Theorem: Stability of Violation

- If running a program on a given heap raises a violation, then it also raises a violation on a heap in which more locations are aliased.
- If running a program produces a result on a given heap, then it also produces a result on a heap with less aliasing.

Theorem: Stability of Violation

- If running a program on a given heap raises a violation, then it also raises a violation on a heap in which more locations are aliased.
- If running a program produces a result on a given heap, then it also produces a result on a heap with less aliasing.

* Unless the program depends on an update to a shared object.

Theorem: Completeness

All accesses through a variable with an access permission contract can only occur via permitted paths.

Implementation

- By transformation of JavaScript code
- Slowdown by a factor of 4–4.4
- Problems: Interfacing with non-transformed code (e.g., library code)
- Large subset of JavaScript supported
- Exceptions: prototype accesses, `with` statements, `eval`
- Goal: implementation in browser / JavaScript engine evades these problems

Evaluation

- Question: effectiveness of access permissions for detecting programming errors
- Method
  - Hand-annotated code run with monitoring
  - Random code modifications
  - Check if modifications detected

Singly-Linked Lists: 6.4% More Errors Detected

	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 %

Richards Benchmark: 13% More Errors Detected

	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%

Effect Inference

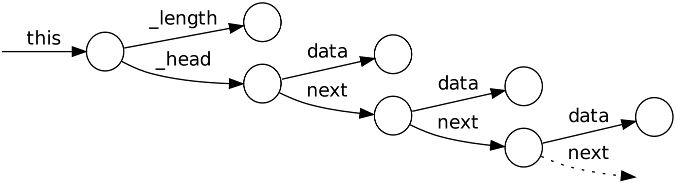
- Where do effects come from?
- For program understand, an automated approach is advantageous.

Sampling

- Program run of a JavaScript program → list of classified access paths
- Example (for add, typically several 1000):

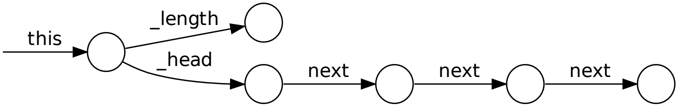
```
R(_head)
R(_head)
R(_head.next)
R(_head.next)
R(_head.next.next)
R(_head.next.next)
R(_head.next.next.next)
W(_head.next.next.next)
R(_length)
W(_length)
```

Example: Structure Derived from Effects



Implementation of Sampling

- Source-to-source transformation of the JavaScript program
- Instruments each property read and write operation
- Annotates objects with path information (anchor and access path)
- Implemented using wrapper objects
- Path set stored in a trie



Hypothesis of the heuristic

Intuition drawn from list and tree datatypes

Hypothesis

Permissions have one of the forms

- $p_1 \ldots p_n.P{*}.q_1 \ldots q_m$
- $p_1 \ldots p_n$

for property names p_i and q_j and a property set P.

Hypothesis of the heuristic

Intuition drawn from list and tree datatypes

Hypothesis

Permissions have one of the forms

- $p_1 \ldots p_n.P*.q_1 \ldots q_m$
- $p_1 \ldots p_n$

for property names p_i and q_j and a property set P.

Need to identify . . .

- common prefixes
- set of middle properties
- common suffixes

Overall inference algorithm

Only for read paths

Input Π^r set of all read paths

— Determine interesting prefixes

$\Pi_0^r \leftarrow \mathrm{Prefixes}(\Pi^r)$

— Infer permissions

$R \leftarrow \mathrm{Permissions}(\mathrm{Reduct}(\Pi_0^r), \Pi^r, sl, sd)$

— Simplify permissions

$\mathrm{Simplify}(R)$

Output $R.@$

Interesting Prefixes

Recall the read paths of the add example:

```
R(_head)
R(_head.next)
R(_head.next.next)
R(_head.next.next.next)
R(_length)
```

Interesting Prefixes

Recall the read paths of the add example:

```
R(_head)
R(_head.next)
R(_head.next.next)
R(_head.next.next.next)
R(_length)
```

The interesting prefixes are `_head` and `_length`. Why?

Interesting Prefixes

Recall the read paths of the add example:

```
R(_head)
R(_head.next)
R(_head.next.next)
R(_head.next.next.next)
R(_length)
```

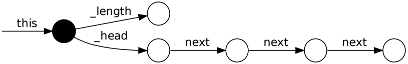
The interesting prefixes are `_head` and `_length`. Why?

A prefix is interesting if . . .

- traversing it *changes* the set of accessible symbols
- extending it *does not change* the set of accessible symbols

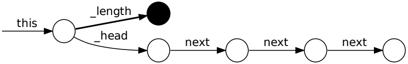
Determining Interesting Prefixes

Prefix: ε



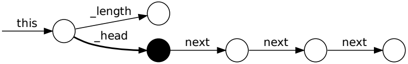
Accessible symbols: `_head`, `_length`, `next`

Prefix: `_length`



Accessible symbols: $\emptyset$

Prefix: `_head`



Accessible symbols `next`

From Interesting Prefix to Permission

From Interesting Prefix to Permission

- Interesting prefixes partition the set of paths Π^r
- For each prefix π determine a permission by extending from the 1-suffixes of the quotient $\pi \setminus \Pi^r$
    - $\texttt{\_length} \setminus \Pi^r = \{\varepsilon\}$
      yields permission _length
    - $\texttt{\_head} \setminus \Pi^r = \{\varepsilon, \texttt{next}, \texttt{next.next}, \texttt{next.next.next}\}$
      yields permissions _head and _head.next$*$.next

From Interesting Prefix to Permission

- Interesting prefixes partition the set of paths Π^r
- For each prefix π determine a permission by extending from the 1-suffixes of the quotient $\pi \setminus \Pi^r$
  - $\texttt{\_length} \setminus \Pi^r = \{\varepsilon\}$
    yields permission _length
  - $\texttt{\_head} \setminus \Pi^r = \{\varepsilon, \texttt{next}, \texttt{next.next}, \texttt{next.next.next}\}$
    yields permissions _head and _head.next$*$.next
- Simplification and making readonly yields _length.@ and _head.next$*$.@

Papers

- Contract-driven Testing of JavaScript Code, TOOLS 2010
- A Heuristic Approach for Computing Effects, TOOLS 2011
- Access Permission Contracts for Scripting Languages, POPL 2012

Conclusions

- JsConTest: Contracts and random testing for JS
- Access permission contracts extend the scope of contracts and monitoring to side effects
- Access permissions fit in with static verification
- Inference of contracts for program understanding

Conclusions

- JsConTest: Contracts and random testing for JS
- Access permission contracts extend the scope of contracts and monitoring to side effects
- Access permissions fit in with static verification
- Inference of contracts for program understanding

Future Work

- Browser instrumentation
- Investigate completeness of inference
- Location-dependent access for security

Conclusions

- JsConTest: Contracts and random testing for JS
- Access permission contracts extend the scope of contracts and monitoring to side effects
- Access permissions fit in with static verification
- Inference of contracts for program understanding

Future Work

- Browser instrumentation
- Investigate completeness of inference
- Location-dependent access for security

`http://proglang.informatik.uni-freiburg.de/jscontest/`

Further Examples

- Layout computation

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

Further Examples

- Layout computation

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

- Objects may be used for keyword parameters:

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

  The parameter object should be read-only:

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

Further Examples

- Layout computation

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

- Objects may be used for keyword parameters:

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

  The parameter object should be read-only:

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

- Observer pattern

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

Further Examples

- Layout computation

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

- Objects may be used for keyword parameters:

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

  The parameter object should be read-only:

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

- Observer pattern

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

- Forbid access to a specific property

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