Recency Types for Dynamically-Typed, Object-Based Languages

Phillip Heidegger, Peter Thiemann

Albert-Ludwigs-Universität Freiburg

15.10.2008

Task: Maintenance

- Finding bugs in JavaScript programs
- Understanding JavaScript programs

JavaScript
Language Properties

Some important features:

1. Weak, dynamic typing
2. Object-based language (no classes, but prototypes)
3. Functions are first class values
4. . . .

Introduction into JavaScript

Example – Objects

```
var ob1 = { a : 5; b : 42 };
var ob2 = new Object();
var x = ob1.a;
var z = ob2.a;

var getA = function () {
  return this.a;
};
ob1.getA = getA;

var x2 = ob1.getA();
```

1. Object literal: $\{lab_1 : e_1; \ldots lab_n : e_n\}$
2. Constructor call using **new**

JavaScript

Example – Prototypes

```
var proto = { test : 5 };
function h(x) {
  this.x = x;
  return this;
};
h.prototype = proto;
var o = new h("Hello!");
alert(o.test + o.x);
```

Function Call after Assignment

Example

```
1 var x = new Object();    // x is an empty object
2 var u = x.f;             // u == undefined
3 x.g = function ... ;
4 x.f();
```

- Wish:

  get a hint that tells us `f` might be undefined in line 4.

Function Call after Assignment

Example

Corrected version:

```
var x = new Object();    // x is an empty object
var u = x.f;              // u == undefined
x.f = function ... ;
x.f();
```

- If the type system is **not** flow-sensitive:
  $\texttt{x.f} : \texttt{undefined} \vee (\tau \rightarrow \tau')$

Types

A short introduction to Union Types

The Types:

$$\tau ::= \texttt{string} \mid \texttt{float} \mid \texttt{undefined} \mid \tau \vee \tau$$

A possible definition:

$$\begin{aligned} \texttt{string} &:= \{s \mid s \text{ is a String}\} \\ \texttt{float} &:= \{f \mid f \text{ is a Float}\} \\ \texttt{undefined} &:= \{\texttt{undefined}\} \\ \tau_1 \vee \tau_2 &:= \tau_1 \cup \tau_2 \end{aligned}$$

Typing of a value:

$$\vdash v : \tau \qquad \text{iff } v \in \tau$$

Types

A short introduction to Union Types

Type rules:

$$\frac{\vdash v : \tau}{\Gamma \vdash v : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad x : \tau \in \Gamma}{\Gamma \vdash x = e : \texttt{undefined}}$$

Example:

```
x = undefined;
x = 5.0;
```

- We may use union types.
- This yields in a typing for the two lines above:

$$[x : \texttt{undefined} \vee \texttt{float}] \vdash x = v : \texttt{undefined}$$

Function Call after Assignment

Example

Corrected version:

```
1 var x = new Object();   // x is an empty object
2 var u = x.f;             // u == undefined
3 x.f = function ... ;
4 x.f();
```

- If the type system is **not** flow-sensitive:
  `x.f` : `undefined` $\vee$ $(\tau \rightarrow \tau')$
- Type system predicts a run-time error for the function call in line 4
- Solution: have `x.f` change type on assignment
- Problem: Unrestricted type change is unsound . . .

Flow Analysis
Standard Approach

- Typical abstraction in a flow analysis:

  Represent object pointer by a set of creation locations

```
var x = new^ℓ Object(); // abstract location ℓ
var u = x.f;            // u == undefined
x.f = function ... ;
x.f () ;
```

- Map each location to an abstract description of an object
- Updates to object have to be suiteable to the abstract description

Flow Analysis

Recency-Based Approach

- Key idea: Distinguish the most recently allocated object from the older ones (at each location)
- Map each location to **two** abstract descriptions, one for the most recent object and one for the older ones

Flow Analysis

Recency-Based Approach

- Observation: The abstract description for the most recent object describes exactly one object!
- Observation: The abstract description for the older objects describes many objects!
- Interpret an update to the most recent object **by an update of the abstract description**
- Interpret an update to an older object by joining abstract descriptions

Semantics

Two heaps:

- H for old objects,
- H_0 for most recent object
- Type of heaps: `reference` $\rightarrow$ `object`
- `references` $::= (q\ell, i)$.
- Type of objects: `string` $\rightarrow$ `value`.

Small step operational semantics:

$$H, H_0, e \longrightarrow H', H_0', e'$$

Selected rules (Read property):

$$\begin{array}{ll} H, H_0, (@\ell, i).a \longrightarrow H, H_0, H_0(\ell, i)(a) & \text{if } (\ell, i) \in \mathrm{dom}(H_0) \\ H, H_0, (^\circ\ell, i).a \longrightarrow H, H_0, H(\ell, i)(a) & \text{if } (\ell, i) \in \mathrm{dom}(H) \end{array}$$

Example for the Semantics

$$[], [], \ \texttt{let}\ x = \texttt{new}^{\ell}\ \texttt{in}\ \texttt{let}\ z = (x.a = 5)\ \texttt{in}\ x.a$$

$$\longrightarrow$$

$$[], [(@\ell, 0) \mapsto \{\}], \ \texttt{let}\ x = (@\ell, 0)\ \texttt{in}\ \texttt{let}\ z = (x.a = 5)\ \texttt{in}\ x.a$$

$$\longrightarrow$$

$$[], [(@\ell, 0) \mapsto \{\}], \ \texttt{let}\ z = ((@\ell, 0).a = 5)\ \texttt{in}\ (@\ell, 0).a$$

Example for the Semantics

$$[], [(@\ell, 0) \mapsto \{\}],\ \texttt{let}\ z = ((@\ell, 0).a = 5)\ \texttt{in}\ (@\ell, 0).a$$

$$\longrightarrow$$

$$[], [(@\ell, 0) \mapsto \{\mathsf{a} \mapsto 5\}],\ \texttt{let}\ z = \texttt{undefined}\ \texttt{in}\ (@\ell, 0).a$$

$$\longrightarrow$$

$$[], [(@\ell, 0) \mapsto \{\mathsf{a} \mapsto 5\}],\ (@\ell, 0).a$$

$$\longrightarrow$$

$$[], [(@\ell, 0) \mapsto \{\mathsf{a} \mapsto 5\}],\ 5$$

Function Calls

Let's assume:

$$\begin{aligned} fb ::= \;& \texttt{let } x = \texttt{new}^{\ell} \texttt{ in} \\ & \texttt{let } y = x \texttt{ in} \\ & \texttt{let } z = (y.a = 5) \texttt{ in} \\ & \quad x.a \end{aligned}$$

$$f ::= \lambda(\_).fb$$

How we evaluate

$$\begin{aligned} & \texttt{let } \_ = f() \texttt{ in} \\ & \quad f() \end{aligned} \qquad ?$$

- We will end up with two objects in the most recent heap!
- When should we move objects into the summary heap?

Moving objects

- Collect a set of abstract locations of objects that are possibly created inside a lambda body
- Move objects with the suitable abstract location from the most recent heap into the summary heap before running the function body

Typing Judgment

$$\Omega, \Sigma, \Gamma \vdash_e e : t \Rightarrow L, \Sigma', \Gamma'$$

- Ω summary environment
- Σ most-recent environment
- Γ type environment
- L effect (set of locations)
- t types

$$\begin{array}{lcl} t & ::= & \texttt{obj}(p) \mid (\Sigma, t \times t) \xrightarrow{L} (\Sigma, t) \mid \top \mid \texttt{undefined} \\ p & ::= & \circ L \mid @\ell \end{array}$$

Object Types

$$\texttt{let } x = \texttt{new}^{\ell} \texttt{ in}$$
$$\texttt{let } y = x \texttt{ in}$$
$$\texttt{let } z = (y.a = 5) \texttt{ in}$$
$$x.a$$

$$\Omega, \Sigma, \Gamma \vdash_e x.a : \texttt{int} \Rightarrow \emptyset, \Sigma, \Gamma$$
$$\Sigma = [\ell \mapsto [a \mapsto \texttt{int}]]$$
$$\Gamma = [x : \texttt{obj}(@\ell), y : \texttt{obj}(@\ell), z : \texttt{undefined}]$$

Object Types with Strong Update

$$\begin{array}{l} \texttt{let } x = \texttt{new}^{\ell} \texttt{ in} \\ \texttt{let } y = x \texttt{ in} \\ \texttt{let } z = (y.a = 5) \texttt{ in} \\ \texttt{let } w = (x.a = \texttt{"crunch"}) \texttt{ in} \\ \quad y.a \end{array}$$

$$\Omega, \Sigma, \Gamma \vdash_e y.a : \texttt{string} \Rightarrow \emptyset, \Sigma, \Gamma$$
$$\Sigma = [\ell \mapsto [a \mapsto \texttt{string}]]$$
$$\Gamma = [x : \texttt{obj}(@\ell), y : \texttt{obj}(@\ell), z : \texttt{undefined}, w : \texttt{undefined}]$$

Object Types with Weak Update

$$
\begin{array}{l}
\quad \texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in} \\
\triangledown^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \texttt{"flush"} \texttt{ in} \\
\triangledown^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true} \texttt{ in} \\
\qquad x.a
\end{array}
$$

$$\Gamma, \Omega, \Sigma \vdash_e \texttt{new}^{\ell} : \texttt{obj}(@\ell) \Rightarrow \{\ell\}, \Gamma, [\ell \mapsto []]$$

$$
\begin{array}{l}
\Omega = [\ell \mapsto [a \mapsto \top]] \\
\Sigma = [] \\
\Gamma = []
\end{array}
$$

Object Types with Weak Update

$$\begin{array}{l} \quad \texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in} \\ \nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \texttt{"flush" in} \\ \nabla^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true in} \\ \qquad x.a \end{array}$$

$$\Gamma, \Omega, \Sigma \vdash_e \ldots : \tau \Rightarrow L, \Gamma', \Sigma'$$

$$\Omega = [\ell \mapsto [a \mapsto \top]]$$
$$\Sigma = [\ell \mapsto []]$$
$$\Gamma = [x : \texttt{obj}(@\ell)]$$

$$\tau = ? \qquad L = ?$$
$$\Gamma' = ? \qquad \Sigma' = ?$$

Object Types with Weak Update

$$\begin{array}{l} \quad\texttt{let}\ x = \texttt{new}^{\ell}\ \texttt{in}\ \texttt{let}\ \_ = x.a = 42\ \texttt{in} \\ \nabla^{\ell}\texttt{let}\ y = \texttt{new}^{\ell}\ \texttt{in}\ \texttt{let}\ \_ = y.a = \texttt{"flush"}\ \texttt{in} \\ \nabla^{\ell}\texttt{let}\ z = \texttt{new}^{\ell}\ \texttt{in}\ \texttt{let}\ \_ = z.a = \texttt{true}\ \texttt{in} \\ \quad\quad x.a \end{array}$$

$$\Gamma, \Omega, \Sigma \vdash_e x.a = 42 : \texttt{undefined} \Rightarrow \emptyset, \Gamma, [\ell \mapsto [a \mapsto \texttt{Float}]]$$

$$\Omega = [\ell \mapsto [a \mapsto \top]]$$
$$\Sigma = [\ell \mapsto []]$$
$$\Gamma = [x : \texttt{obj}(@\ell)]$$

Object Types with Weak Update

$$\begin{array}{l} \quad \texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in} \\ \nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \texttt{"flush" in} \\ \nabla^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true in} \\ \quad\quad x.a \end{array}$$

$$\Gamma, \Omega, \Sigma \vdash_e \nabla^{\ell} e : \tau \Rightarrow L, \Gamma', []$$

$$\Omega = [\ell \mapsto [a \mapsto \top]]$$
$$\Sigma = [\ell \mapsto [a \mapsto \texttt{Float}]]$$
$$\Gamma = [x : \texttt{obj}(@\ell)]$$

$$\Gamma' = [x : \texttt{obj}(^{\circ}\ell)]$$

Object Types with Weak Update

$$
\begin{array}{l}
\quad\texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in} \\
\nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \texttt{"flush" in} \\
\nabla^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true in} \\
\quad\quad x.a
\end{array}
$$

$$\Gamma, \Omega, \Sigma \vdash_e \ldots : \tau \Rightarrow L, \Gamma', []$$

$$
\begin{array}{l}
\Omega = [\ell \mapsto [a \mapsto \top]] \\
\Sigma = [] \\
\Gamma = [x : \texttt{obj}(^{\circ}\ell)]
\end{array}
$$

Object Types with Weak Update

$$
\begin{array}{l}
\texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in} \\
\nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \texttt{"flush" in} \\
\nabla^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true in} \\
\quad x.a
\end{array}
$$

$$
\begin{array}{l}
\Omega, \Sigma, \Gamma \vdash_e x.a : \top \Rightarrow \emptyset, \Sigma, \Gamma \\
\Omega = [\ell \mapsto [a \mapsto \top]] \\
\Sigma = [\ell \mapsto [a \mapsto \texttt{Bool}]] \\
\Gamma = [x : \texttt{obj}(\circ\ell), y : \texttt{obj}(\circ\ell), z : \texttt{obj}(@\ell)]
\end{array}
$$

Mask Expressions

A mask expression $\nabla^L e$

- not written by the programmer
- inserted in an elaboration phase
- syntactic marker for moving objects from the most-recent heap to the summary heap

Where to Mask

Elaboration applies a mask to each `let` that

- directly encloses a function call, or
- directly encloses a method call

The mask label is inferred.

Conclusion

- Recency can be modeled with a type system
- Recency types partition the lifetime of an object
  - initialization phase (most recent object, strong updates)
  - summary phase (old object, weak updates)

  Intuition: most objects are initialized once and then changed rarely
- Recency types deal well with prototypes
  - prototypes are often singleton objects
  - precise and imprecise pointers can be arbitrarily nested
- Implementation: up next

Attention

Thank you for your attention!