Recency Types for JavaScript

Phillip Heidegger, Peter Thiemann

University of Freiburg

08.06.2009

Motivation

Create a static program analysis to:

- Find bugs in JavaScript programs
- Understand JavaScript programs

- Specify this analysis as a type system

Properties of JavaScript

Some important features:

- Object-based language (no classes, but prototypes)
- Functions are first class values
- Weak, dynamic typing

- The combination of these properties makes a static analysis a challenge

Example:
Type State Required

```
1 var x = new Object();  // x is an empty object
2 x.a = "Hello";         // x.a : string
3 x.a = function (){};   // x.a : () → undefined
4 x.a();
```

- Flow insensitive type systems will reject line 4.
- Solution: Allow type of `x.a` to change
- Problem: Unrestricted type change of `x.a` is unsound

Our Solution (1/2): Recency

- **Observation**: In a dynamically typed language, objects have an initialization phase where updates are needed, but afterwards type remain stable (supported by recent study by Vitek et al)

- **Key idea:** At each location, distinguish the abstraction of the most recently allocated object from the older ones.

Dynamic Semantics

```
function foo() {
  var x = new^k Object();
  x.a = "Hallo";
  x.a = function () {};
  return x;
}
var y = foo();
var z = foo();
```

Summary Heap

Most Recent Heap

z,k { a : function () {} }

How to move the objects?

- Separate between the movement and the creation
- Introduce a new expression: **`MASK`**^{k}
- **`new`**^{k} **`Constr()`** creates new objects in most recent heap
- **`MASK`**^{k} moves objects from most recent heap to summary heap

Modified Example

```
function foo() {
  MASK^k;
  var x = new^k Object();
  x.a = “Hallo”;
  x.a = function () {};
  return x;
}
var y = foo();
var z = foo();
```

- **MASK**^{k} moves objects
- Automatic insertion of **MASK**^{k}

Our Solution (2/2): Flow Analysis

- Typical abstraction in a flow analysis:

  Represent object pointer by an abstract location!

```
var x = new^k Object();
```

- Mark each `new` expression with an abstract location
- Object types: `obj(@k)` or `obj(~k)`
- Abstract heap maps each abstract location to an object description

Static Typing: Example 1/2

```
MASK^k;
var x = new^k Object();   // x : obj(@k)
x.a = "Hello";
x.a = function () {};
MASK^k;                   // x : obj(~k)
var y = new^k Object();   // y : obj(@k), x : obj(~k)
```

Summary heap type

Most recent heap type

`~k` { a : () → undefined }

Before

After

`@k` { a : () → undefined }

Static Typing: Example 2/2

```
function foo() {              // foo: () → obj(@k)
  MASK^k;                     // most r. Heap Input
  var x = new^k Object();     //              ?
  x.a = “Hallo”;              // most r. Heap Output
  x.a = function () {};       // @k: {a: () → undefined}
  return x;
}
var y = foo();   // y : obj(@k)
var z = foo();   // y : obj(~k),   z : obj(@k)
var v = foo();   // y,z : obj(~k), v : obj(@k)
```

Static Typing: Example 2/2

```
function foo() {             // foo: () → obj(@k)
  MASK^k;                    // most r. Heap Input
  var x = new^k Object();    //          ---
  x.a = "Hallo";             // most r. Heap Output
  x.a = function () {};      // @k: {a: () → undefined}
  return x;
}
MASK^k;
var y = foo();               // foo: () → obj(@k)
MASK^k;
var z = foo();               // foo: () → obj(@k)
MASK^k;
var v = foo();               // foo: () → obj(@k)
```

Prototypes

- JavaScript's mechanism for inheritance
- Assumption: prototypes are singleton objects
  - Stay most recent
  - Strong updates during the whole program execution
  - Gives flexibility to objects in the summary heap

Related Work

- Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. SAS 2006.
- Simon Holm Jensen, Anders Möller, and Peter Thiemann. Type Analysis for JavaScript. SAS 2009.

Related Work

- Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. ECOOP 2005.
  - We can type each program they can type
  - We allow type changes (for most recent objects)
  - We can deal with prototypes

Related Work

- Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. ESOP 2000.
- David Walker and Greg Morrisett. Alias types for recursive data structures. TIC 2000.

Alias Types

- Type checking
- unroll/unpack operations

Recency Types

- Type inference
- No movement from summary into most recent heap

Conclusion

- Recency and flow sensitivity = Sweet Spot
  - Strong updates during initialization
  - Weak updates after initialization
  - → Strong updates on prototypes
- Type inference (no annotations)
- Future Work
  - Real world programs?
  - Abstract over locations, conditionals, sharing, ...

Thank you for your attention !

Questions?

Typing the Mask Expression

- Typing of a mask expression **MASK**^{k} is safe, if the heap descriptions for the summary heap is a super type of the most recent heap description for the abstract location k.

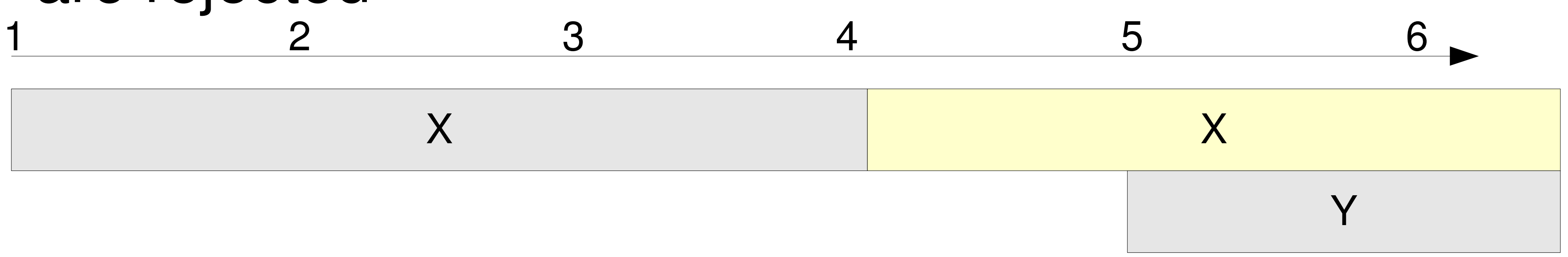
$$\Omega(k) :> \Sigma(k)$$

➔ Typing of the mask expression depends on the most recent heap description

Life Time of an Object

```
1 MASK^k;
2 var x = new^k Object();   // x : obj(@k)
3 x.a = "Hello";
4 x.a = function () {};
5 MASK^k;                   // x : obs(~k)
6 var y = new^k Object();   // y : obj(@k), x : obs(~k)
```

- The Object x, (Line 1) is precise from Line 1 to Line 4.
- After line 4 it becomes old. Since that strong updates are rejected



Static Type System

- Make use of the distinction
- Type Judgment:

$$\Gamma,\Omega,\Sigma\ `_{e}\ e : t\ \ )\ \Sigma',\Gamma'$$

  - Γ : Type Environment

    variables → types
  - Ω : models the summary heap

    abstr. locations → object types
  - Σ : models the most recent heap

    abstr. locations → object types

Related Work

- the precise type **`obj(@l)`** expresses that all variables of this type refer to the same object.
- the imprecise type **`obj(~L)`** express may-alias information
  - John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In ECOOP '01, London, UK, 2001. Springer-Verlag.
  - Rita Z. Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In Proc. 1995 ACM Symp. POPL, San Francisco, CA, USA, January 1995. ACM Press.