Types and Analysis for Scripting Languages

Peter Thiemann

Universität Freiburg, Germany

Mini Lecture, Tallinn, Estland, 25.-27.2.2008

Introduction

Ultra-Brief JavaScript Tutorial

Dynamic Typing and Soft Typing
Scripting Languages

- Lightweight programming languages evolved from command languages
- Lightweight data structures hashmap (object), strings
- Lightweight syntax familiar, no semicolon, (often not well specified), 
- Lightweight typing dynamic, weak, duck typing
- Lightweight metaprogramming
- Lightweight implementation interpreted, few tools
Uses of Scripting Languages

- Glue language for components
- Configuration and automation of complex software
graphics, visualization, office software, web frameworks
- Embedded languages
- Web scripting
  - server-side
  - client-side
- Examples: php, perl, ruby, lua, python, javascript, vb-script, scheme, emacs-lisp, awk, sh, tcl/tk, groovy . . .
JavaScript, a Typical Scripting Language

- Initially developed by Netscape’s Brendan Eich
- Standardized as ECMAScript (ECMA-262 Edition 3)
- Application areas (scripting targets)
  - client-side web scripting (dynamic HTML, SVG, XUL)
  - server-side scripting (Whitebeam, Helma, Cocoon, iPlanet)
  - animation scripting (diablo, dim3, k3d)
  - and many more
JavaScript, Technically

- Java-style syntax
- Object-based imperative language
  - no classes, but prototype concept
  - objects are hashtables
- First-class functions
  - a functional language
- Weak, dynamic type system

**Slogan:** Any type can be converted to any other reasonable type

```javascript
node.onmouseout = function (ev) {
    init();
    state++;
    node.className = "highlight-" + state;
    ev.stopPropagation();
};
```
Problems with JavaScript

Symptomatic for other scripting languages

- No module system
  - No namespace management
  - No interface descriptions

- No application specific datatypes
  primitive datatypes, strings, hashtables

- Type conversions are sometimes surprising
  “A scripting language should never throw an exception [the script should just continue]” (Rob Pike, Google)

- Few development tools (debugger)

⇒ Limited to small applications
Specific Problems with JavaScript

- Most popular applications
  - client-side scripting
  - AJAX
- Dynamic modification of page content via DOM interface
  - DOM = document object model
  - W3C standard interface for accessing and modifying XML
  - Mainly used in web browsers
Specific Problems with JavaScript

- Most popular applications
  - client-side scripting
  - AJAX
- Dynamic modification of page content via DOM interface
  - DOM = document object model
  - W3C standard interface for accessing and modifying XML
  - Mainly used in web browsers
- Incompatible DOM implementations in Web browsers
  ⇒ programming recipes instead of techniques
Can You Write Reliable Programs in JavaScript?

- Struggle with the lack of e.g. a module system
  - Ad-hoc structuring of large programs
  - Naming conventions
  - Working in a team
- Work around DOM incompatibilities
  - Use existing JavaScript frameworks (widgets, networking)
  - Frameworks are also incompatible
- Wonder about unexpected results
- Instance of Dick Gabriel’s “Worse is Better” Claim
Excursion: MIT/Stanford Style of Design
MIT Approach (Dick Gabriel)

Simplicity  the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.

Correctness  the design must be correct in all observable aspects. Incorrectness is simply not allowed.

Consistency  the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.

Completeness  the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.
Excursion: Worse-Is-Better Design Philosophy
New Jersey Approach (Dick Gabriel)

Simplicity  the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.

Correctness  the design must be correct in all observable aspects. It is slightly better to be simple than correct.

Consistency  the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.

Completeness  the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of other qualities, but not at the expense of correctness.
Introduction

Ultra-Brief JavaScript Tutorial

Dynamic Typing and Soft Typing
An Ultra-Brief JavaScript Tutorial

Rule 1:
JavaScript is object-based. An object is a hash table that maps named properties to values.
An Ultra-Brief JavaScript Tutorial

Rule 1:
JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:
Every value has a type. For most reasonable combinations, values can be converted from one type to another type.
Rule 1:
JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:
Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

Rule 3:
Types include `null`, `boolean`, `number`, `string`, `object`, and `function`. 
Rule 1:
JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:
Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

Rule 3:
Types include null, boolean, number, string, object, and function.

Rule 4:
‘Undefined’ is a value (and a type).
Some Quick Questions

Let’s define an object `obj`:

```javascript
js> var obj = { x: 1 }
```

What are the values/outputs of

- `obj.x`
- `obj.y`
- `print(obj.y)`
- `obj.y.z`
Answers

```
js> var obj = {x:1}
js> obj.x
1
js> obj.y
js> print(obj.y)
undefined
js> obj.y.z
js: "<stdin>", line 12: uncaught JavaScript exception:
    ConversionError: The undefined value has no properties.
    (<stdin>; line 12)
```
Rule 5:
An object is really a dynamic mapping from strings to values.

```javascript
js> var x = "x"
js> obj[x]
1
js> obj.undefined = "gotcha"
gotcha
js> obj[obj.y]
```

What is the effect/result of the last expression?
Rule 5:
An object is really a dynamic mapping from strings to values.

```javascript
js> var x = "x"
js> obj[x]
1
js> obj.undefined = "gotcha"
gotcha
js> obj[obj.y]
    == obj[undefined]
    == obj["undefined"]
    == obj.undefined
    == "gotcha"
```
Recall Rule 2:
Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

```
js> var a = 17
js> a.x = 42
42
js> a.x
```

What is the effect/result of the last expression?
Weak, Dynamic Types in JavaScript III

Wrapper objects for numbers

```javascript
js> m = new Number (17); n = new Number (4)
js> m+n
21
```
Wrapper objects for numbers

```javascript
js> m = new Number (17); n = new Number (4)
js> m+n
21
```

Wrapper objects for booleans

```javascript
js> flag = new Bool(false);
js> result = flag ? true : false;
```

What is the value of `result`?
Rule 6:
Functions are first-class, but behave differently when used as methods or as constructors.

```javascript
js> function f () { return this.x }
js> f()
x
js> obj.f = f
function f() { return this.x; }
js> obj.f()
1
js> new f()
[object Object]
```
js> obju = { u : {} }  
[object Object]

js> objv = { v : {} }  
[object Object]

js> print(obju.u)
undefined

js> print(objv.u)
undefined
Rule 7:
The `with` construct puts its argument object on top of the current environment stack.

```javascript
js> u = "defined"
defined
js> with (obju) print(u)
undefined
js> with (objv) print(v)
defined
```
Rule 8:
The `for` construct has an `in` operator to range over all defined indexes.

```javascript
js> for (i in obju) print(i)
  u
js> for (i in objv) print(i)
  v
js> delete objv.v
  true
js> for (i in objv) print(i)
  js> delete objv.v
  true
```
Strings as Data Structures

“Semantics buried in strings is the ultimate evil”

(Erik Meijer, 2004)
Strings as Data Structures I
HTML+XML

elements[i].innerHTML = 
' <div nowrap="nowrap">' +
elements[i].innerHTML +
' </div> ';
...
area = document.getElementById("waitMsg");
area.innerHTML = 
" <em> <font color=red> Please wait while 
the submission is being uploaded. 
</font> </em> " ;
...
var tT='+++ some text +++';
t=document.getElementById('ticker');
t.innerHTML=' <div> '+tT+' </div><div>'+tT+'</div>';
Typical pattern in JavaScript, Php, Servlets, . . . :

Create HTML by concatenating and printing strings

Problems

- well-formedness of HTML/XML not guaranteed
  (i.e., the document may not be a tree)

- validity of HTML/XML not guaranteed
  (i.e., the document may not adhere to a DTD or XMLSchema specification)

Consequence: browsers and XML processors may choke
Strings as Data Structures III
Solutions for HTML+XML

- Partial approach: E4X
  - JavaScript dialect
  - XML literals and XPath processing
  - but implemented in terms of strings
- Full solution: DOM
  - Fairly clumsy to use
- Wait for Meijer’s law:
  “Any sufficiently heavily used API or common programming pattern will eventually evolve into a first-class language feature or construct”
Strings as Data Structures IV
Further string-embedded languages

- XPath
- SQL
  - not just in scripting languages!
  - in JDBC:
    ```java
    ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
    ```
- Also addressed by language extenders and grammar systems
  - (not that widely used)
State of Affairs

Theses

- SLs are used in ad-hoc ways by choice
- SL programmers want the freedom given by data structures encoded in strings
- SL programmers want dynamic typing and conversions
- SL programmers write large programs (and need assurance for them)
State of Affairs II

Goal
- Keep today’s usage patterns of SLs
- Enable maintenance
- Improve assurance if needed

Means: Typing and static analysis
- to detect problems with dynamic typing (errors, unwanted conversions)
- to detect malformed strings
- to address problems with incompletely specified APIs

Inspiration
- Soft typing
- Flow analysis
- Work on grammars and parsing
Introduction

Ultra-Brief JavaScript Tutorial

Dynamic Typing and Soft Typing
Soft Typing


- Problem statement
  - Static type checking for a dynamically typed language (Scheme)
  - Untyped programs should not be rejected
  - Typings results should be easy to interpret
  - Type inference should be efficient
Example of Soft Typing

(define flatten
  (lambda (l)
    (cond
      ((null? l)
        '())
      ((pair? l)
        (append (flatten (car l))
                (flatten (cdr l))))
      (else
        (list l))))
(define a '(1 (2) 3))
(define b (flatten a))
Example of Soft Typing

Results

```
flattenn

(rec ((Y1 (+ nil (cons Y1 Y1) X1)))
    (Y1 -> (list (+ (not nil) (not cons) X1)))))

a

(cons num (cons (cons num nil) (cons num nil)))

b

(list num)
```
Example of Soft Typing

Explanation

- **num** type of numbers
- **nil** type of empty list
- **(cons X Y)** type of a pair
- **(+ X Y)** union type
- **(not nil)** type which does not include the empty list
- **(not cons)** type which does not include a pair
- **(list Z)** stands for the recursive type
  
  (rec ((Y (+ nil (cons Z Y)))) Y)
Type Language

\[ T ::= (+ P_1 \ldots P_n) | (+ P_1 \ldots P_n X) \]
\[ P ::= \text{num} | \text{nil} | (\text{cons } T T) | (T_1 \ldots T_n \rightarrow T) | N \]
\[ N ::= (\text{not num}) | (\text{not nil}) | (\text{not cons}) | (\text{not } \rightarrow) \]
\[ R ::= (\text{rec } ((X_1 T_1) \ldots (X_n T_n)) T) | T \]

- Restrictions
  - Each tag must be used at most once in each union. (discriminative sum type, see Henglein and Rehof)
  - The same set of tags must always precede a particular type variable.
    - The \( N \) serve as place holders for absent tags.
  - Recursive Types \( R \) via first-order recursive recursive equations.
Types of Well-known Scheme Functions

map:

\[(X_1 \rightarrow X_2) \ (\text{list } X_1) \rightarrow (\text{list } X_2)\]

member:

\[(X_1 \ (\text{list } X_2) \rightarrow (+ \ \text{false} \ (\text{cons } X_2 \ (\text{list } X_2)))\]

read:

\[(\text{rec } ((Y_1 (+ \ \text{num} \ \text{nil} \ \text{\ldots} \ (\text{cons } Y_1 Y_1))))
\rightarrow (+ \ \text{eof} \ \text{num} \ \text{nil} \ \text{\ldots} \ (\text{cons } Y_1 Y_1))))\]

lastpair:

\[(\text{rec } ((Y_1 (+ \ (\text{cons } X_1 Y_1) X_2))))
\ ((\text{cons } X_1 Y_1) \rightarrow (\text{cons } X_1 (+ \ (\text{not cons}) X_2))))\]
Core Calculus

\[(\text{Exp}) \quad e \ ::= \ v \mid (\text{ap} \ e \ e) \mid (\text{CHECK-ap} \ e \ e) \mid (\text{let} \ ([x \ e]) \ e)\]

\[(\text{Val}) \quad v \ ::= \ c \mid x \mid (\text{lambda} \ (x) \ e)\]

where

- $x \in Id$ identifiers
- $c \in Const$ constants (basic constants and primitive operations)
- checked and unchecked primitives
Types for the Core Calculus

Inspired by domain equation for data

\[ \mathcal{D} = \mathcal{D}_{num} \oplus \mathcal{D}_{true} \oplus \mathcal{D}_{false} \oplus \mathcal{D}_{nil} \oplus (\mathcal{D} \otimes \mathcal{D}) \oplus [\mathcal{D} \circ \to \mathcal{D}]_{\perp} \]

\[ \mathcal{D}_{num} = \{ \ldots, -1, 0, 1, 2, \ldots \}_{\perp} \]

\[ \mathcal{D}_{true} = \{ \#t \}_{\perp} \]

\[ \mathcal{D}_{false} = \{ \#f \}_{\perp} \]

\[ \mathcal{D}_{nil} = \{ nil \}_{\perp} \]
Types for the Core Calculus II

Type language

\[
\sigma, \tau \ ::= \ \kappa_1^{f_1} \vec{\sigma}_1 \cup \ldots \cup \kappa_n^{f_n} \vec{\sigma}_n \cup (\alpha \mid \emptyset) \\
\kappa \in \text{Tag} = \{\text{num, true, false, nil, cons, \text{-}>}\} \\
f \ ::= \ + \mid - \mid \varphi
\]

- Types must be *tidy*: each tag must not occur more than once in a union (cf. Rémy’s and Wand’s row types)
- Types may be recursive (using \(\mu\) notation)
Types for the Core Calculus III

Examples

\[ \text{num}^+ \cup \emptyset \]

\[ \text{num}^+ \cup \text{nil}^+ \cup \emptyset \]

\[ \text{num}^+ \cup \text{nil}^- \cup \alpha \]

\[ (\alpha \rightarrow^+ (\text{true}^+ \cup \text{false}^+ \cup \emptyset)) \cup \emptyset \]

➤ (types get big quickly)
At the top level, types can be abstracted over
- type variables
- flag variables

Type schemes

\[ \forall \vec{\alpha} \vec{\varphi}. \tau \]

Stands for a set of substitution instances
- a substitution for \( \alpha \) must not destroy tidyness
- a substitution for \( \varphi \) must be in \{ +, −, \( \varphi' \) \}
Types for the Core Calculus V
Polymorphism for Encoding Subtyping

- \( \forall \alpha. \text{num}^+ \cup \alpha \) can be instantiated to any type that includes \text{num}
  \[
  \begin{align*}
  \text{num}^+ \cup \emptyset \\
  \text{num}^+ \cup \text{true}^+ \cup \emptyset \\
  \text{num}^+ \cup \text{true}^+ \cup \text{false}^+ \cup \emptyset
  \end{align*}
  \]

- \( \forall \varphi_1 \varphi_2. \text{num}^{\varphi_1} \cup \text{nil}^{\varphi_2} \cup \emptyset \) can be instantiated to any type that is contained in \( \text{num}^+ \cup \text{nil}^+ \cup \emptyset \)
  \[
  \begin{align*}
  \text{num}^+ \cup \text{nil}^+ \cup \emptyset \\
  \text{num}^+ \cup \text{nil}^{-} \cup \emptyset \\
  \text{num}^{-} \cup \text{nil}^+ \cup \emptyset \\
  \text{num}^{-} \cup \text{nil}^{-} \cup \emptyset
  \end{align*}
  \]
Types for the Core Calculus VI
Types of Checked and Unchecked Primitives

\[ \text{.TypeOf}(0) = \forall \alpha. \]  
\[ \text{num}^+ \cup \alpha \]

\[ \text{.TypeOf}(\text{add1}) = \forall \alpha_1 \alpha_2 \varphi. \]  
\[ ((\text{num}^\varphi \cup \emptyset) \rightarrow^+ (\text{num}^+ \cup \alpha_1)) \cup \alpha_2 \]

\[ \text{.TypeOf}(\text{number}?) = \forall \alpha_1 \alpha_2 \alpha_3. \]  
\[ (\alpha_1 \rightarrow^+ (\text{true}^+ \cup \text{false}^+ \cup \alpha_2)) \cup \alpha_3 \]

\[ \text{.TypeOf}(\text{CK-add1}) = \forall \alpha_1 \alpha_2 \alpha_3 \varphi. \]  
\[ ((\text{num}^\varphi \cup \alpha_3) \rightarrow^+ (\text{num}^+ \cup \alpha_1)) \cup \alpha_2 \]
Typing Rules

(const) \[ \frac{\tau \prec \text{TypeOf}(c)}{A \vdash c : \tau} \]

(var) \[ \frac{\tau \prec A(x)}{A \vdash x : \tau} \]

(ap) \[ \frac{A \vdash e_1 : (\tau_2 \rightarrow^f \tau_1) \cup \emptyset \quad A \vdash e_2 : \tau_2}{A \vdash (\text{ap} \ e_1 \ e_2) : \tau_1} \]

(Cap) \[ \frac{A \vdash e_1 : (\tau_2 \rightarrow^f \tau_1) \cup \tau_3 \quad A \vdash e_2 : \tau_2}{A \vdash (\text{CHECK-ap} \ e_1 \ e_2) : \tau_1} \]

(lam) \[ \frac{A, x : \tau_2 \vdash e : \tau_1}{A \vdash (\text{lambda} \ (x) \ e) : (\tau_2 \rightarrow^+ \tau_1) \cup \tau_3} \]

(let) \[ \frac{A \vdash e_1 : \tau_1 \quad A, x : \forall \bar{\alpha} \bar{\phi} \tau_1 \vdash e_2 : \tau_2 \quad \bar{\alpha} \bar{\phi} = fV(\tau_1) \setminus fV(A)}{A \vdash (\text{let} \ ([x \ e_1]) \ e_2) : \tau_2} \]
The operational semantics is standard (small-step).

- Checked applications reduce to an error term
- Unchecked applications get stuck

Theorem (Type Soundness)

If $\emptyset \vdash e : \tau$ then either $e$ diverges or $e \rightarrow^* \text{error}$ or $e \rightarrow^* v$ with $\emptyset \vdash v : \tau$. 
Stepping up to Soft Typing

- The type system for the core calculus is type sound
- But it rejects some meaningful programs
- Against the intention of soft typing!
Stepping up to Soft Typing

- The type system for the core calculus is type sound
- But it rejects some meaningful programs
- Against the intention of soft typing!
- Soft typing should accept all programs
- Insert run-time checks if static type safety cannot be shown
Absent Variables

- Function $SoftTypeOf$ is identical to $TypeOf$ for checked primitives.
- For unchecked primitives it converts the $TypeOf$ type by replacing - flags and $\emptyset$ types to absent flag and type variables ($\bar{\nu}$ in the rules).
- Typing rules check absent variables for emptiness and chose checked or unchecked versions as appropriate.

**Example**

$$TypeOf(\text{add1}) = \forall \alpha_1 \alpha_2 \varphi. \quad (((\text{num}^\varphi \cup \emptyset) \rightarrow^+ (\text{num}^+ \cup \alpha_1)) \cup \alpha_2)$$

$$SoftTypeOf(\text{add1}) = \forall \alpha_1 \alpha_2 \bar{\alpha}_3 \varphi. \quad (((\text{num}^\varphi \cup \bar{\alpha}_3) \rightarrow^+ (\text{num}^+ \cup \alpha_1)) \cup \alpha_2)$$
Soft Typing Transformation Rules

\[(\text{const})\] \[\frac{\tau \prec_S \text{TypeOf}(c)}{A \vdash_s c \Rightarrow (\text{empty}\{S\nu \mid \nu \in \text{dom}(S)\}) \rightarrow c, \text{CHECK-} c) : \tau}\]

\[(\text{var})\] \[\frac{\tau \prec A(x)}{A \vdash_s x \Rightarrow x : \tau}\]

\[(\text{ap})\] \[\frac{A \vdash_s e_1 \Rightarrow e'_1 : (\tau_2 \rightarrow f \tau_1) \cup \tilde{\tau}_3 \quad A \vdash_s e_2 \Rightarrow e'_2 : \tau_2}{A \vdash_s (\text{ap} e_1 e_2) \Rightarrow (\text{empty}\{\tilde{\tau}_3\} \rightarrow (\text{ap} e'_1 e'_2)), (\text{CHECK-ap} e'_1 e'_2)) : \tau_1}\]

\[(\text{Cap})\] \[\frac{A \vdash_s e_1 \Rightarrow e'_1 : (\tau_2 \rightarrow f \tau_1) \cup \tau_3 \quad A \vdash_s e_2 \Rightarrow e'_2 : \tau_2}{A \vdash_s (\text{CHECK-ap} e_1 e_2) \Rightarrow (\text{CHECK-ap} e'_1 e'_2) : \tau_1}\]

\[(\text{lam})\] \[\frac{A, x : \tau_2 \vdash_s e \Rightarrow e' : \tau_1}{A \vdash_s (\text{lambda} (x) e) \Rightarrow (\text{lambda} (x) e') : (\tau_2 \rightarrow + \tau_1) \cup \tau_3}\]

\[(\text{let})\] \[\frac{A \vdash_s e_1 \Rightarrow e'_1 : \tau_1 \quad A, x : \forall \tilde{\alpha} \tilde{\phi} \tau_1 \vdash_s e_2 \Rightarrow e'_2 : \tau_2}{\tilde{\alpha} \tilde{\phi} = (fv(\tau_1) \setminus fv(A)) \setminus \text{AbsentVar} \quad A \vdash_s (\text{let} ([x e_1]) e_2) \Rightarrow (\text{let} ([x e'_1]) e'_2) : \tau_2}\]
Related Work

- Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. FPCA 1995.
- Didier Rémy’s work on row types (further developed by Pottier).