Types and Analysis for Scripting Languages

Peter Thiemann

Universität Freiburg, Germany

Mini Lecture, Tallinn, Estland, 25.-27.2.2008

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

```
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)

$\Rightarrow$ 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 browers

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 browers
- Incompatible DOM implementations in Web browsers

$\Rightarrow$ 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***

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.

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

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 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`:

```
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 excepti
 ConversionError: The undefined value has no proper
 (<stdin>; line 12)
```

Weak, Dynamic Types in JavaScript II

Rule 5:

An object is really a dynamic mapping from strings to values.

```
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?

Weak, Dynamic Types in JavaScript II

Rule 5:

An object is really a dynamic mapping from strings to values.

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

Weak, Dynamic Types in JavaScript III

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

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

Weak, Dynamic Types in JavaScript III

Wrapper objects for numbers

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

Wrapper objects for booleans

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

What is the value of `result`?

Weak, Dynamic Types in JavaScript IV

Rule 6:

Functions are first-class, but behave differently when used as methods or as constructors.

```
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]
```

Distinguishing Absence and Undefinedness I

```
js> obju = { u : {}.xx }
[object Object]
js> objv = { v : {}.xx }
[object Object]
js> print(obju.u)
undefined
js> print(objv.u)
undefined
```

Distinguishing Absence and Undefinedness II

Rule 7:

The `with` construct puts its argument object on top of the current environment stack.

```
js> u = "defined"
defined
js> with (obju) print(u)
undefined
js> with (objv) print(v)
defined
```

Distinguishing Absence and Undefinedness III

Rule 8:

The `for` construct has an `in` operator to range over all defined indexes.

```
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>';
...
```

Strings as Data Structures II
HTML+XML

- 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 XMLScheme 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:

```
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

Soft Typing

- Example: Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. Toplas 19:1(87-152), 1997.
- 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

flatten

```
(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

$$
\begin{array}{lcl}
T & ::= & (+\ P_1\ \ldots\ P_n) \mid (+\ P_1\ \ldots\ P_n\ X) \\
P & ::= & \texttt{num} \mid \texttt{nil} \mid (\texttt{cons}\ T\ T) \mid (T_1\ \ldots\ T_n\ \texttt{->}\ T) \mid N \\
N & ::= & (\texttt{not num}) \mid (\texttt{not nil}) \mid (\texttt{not cons}) \mid (\texttt{not ->}) \\
R & ::= & (\texttt{rec}\ ((X_1\ T_1) \ldots (X_n\ T_n))\ T) \mid T
\end{array}
$$

- 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 equations.

Types of Well-known Scheme Functions

map:

```
((X1 -> X2) (list X1) -> (list X2))
```

member:

```
(X1 (list X2) -> (+ false (cons X2 (list X2))))
```

read:

```
(rec ((Y1 (+ num nil \dots (cons Y1 Y1))))
  (-> (+ eof num nil \dots (cons Y1 Y1))))
```

lastpair:

```
(rec ((Y1 (+ (cons X1 Y1) X2)))
  ((cons X1 Y1) -> (cons X1 (+ (not cons) X2))))
```

Core Calculus

$$
\begin{array}{llll}
\textit{(Exp)} & e & ::= & v \mid (\textbf{ap}\ e\ e) \mid (\textbf{CHECK-ap}\ e\ e) \mid (\textbf{let}\ ([x\ e])\ e) \\
\textit{(Val)} & v & ::= & c \mid x \mid (\textbf{lambda}\ (x)\ e)
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{D} & = & \mathcal{D}_{num} \oplus \mathcal{D}_{true} \oplus \mathcal{D}_{false} \oplus \mathcal{D}_{nil} \oplus (\mathcal{D} \otimes \mathcal{D}) \oplus [\mathcal{D} \circ\!\!\longrightarrow \mathcal{D}]_{\perp} \\
\mathcal{D}_{n}um & = & \{\ldots, -1, 0, 1, 2, \ldots\}_{\perp} \\
\mathcal{D}_{true} & = & \{\#\mathtt{t}\}_{\perp} \\
\mathcal{D}_{false} & = & \{\#\mathtt{f}\}_{\perp} \\
\mathcal{D}_{nil} & = & \{nil\}_{\perp}
\end{array}
$$

Types for the Core Calculus II

Type language

$$
\begin{array}{lcl}
\sigma, \tau & ::= & \kappa_1^{f_1}\vec{\sigma_1} \cup \ldots \kappa_n^{f_n}\vec{\sigma_n} \cup (\alpha \mid \emptyset) \\
\kappa & \in & \mathit{Tag} = \{\mathtt{num}, \mathtt{true}, \mathtt{false}, \mathtt{nil}, \mathtt{cons}, \mathtt{->}\} \\
f & ::= & + \mid - \mid \varphi
\end{array}
$$

- 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 μ notation)

Types for the Core Calculus III

Examples

$$\texttt{num}^+ \cup \emptyset$$

$$\texttt{num}^+ \cup \texttt{nil}^+ \cup \emptyset$$

$$\texttt{num}^+ \cup \texttt{nil}^- \cup \alpha$$

$$(\alpha \texttt{->}^+ (\texttt{true}^+ \cup \texttt{false}^+ \cup \emptyset)) \cup \emptyset$$

- (types get big quickly)

Types for the Core Calculus IV

Type Schemes

- 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 α must not destroy tidyness
  - a substitution for φ must be in $\{+, -, \varphi'\}$

Types for the Core Calculus V

Polymorphism for Encoding Subtyping

- $\forall\alpha.\mathtt{num}^+ \cup \alpha$ can be instantiated to any type that *includes* $\mathtt{num}$

$$\begin{array}{l}\mathtt{num}^+ \cup \emptyset \\ \mathtt{num}^+ \cup \mathtt{true}^+ \cup \emptyset \\ \mathtt{num}^+ \cup \mathtt{true}^+ \cup \mathtt{false}^+ \cup \emptyset\end{array}$$

- $\forall\varphi_1\varphi_2.\mathtt{num}^{\varphi_1} \cup \mathtt{nil}^{\varphi_2} \cup \emptyset$ can be instantiated to any type that is *contained* in $\mathtt{num}^+ \cup \mathtt{nil}^+ \cup \emptyset$

$$\begin{array}{l}\mathtt{num}^+ \cup \mathtt{nil}^+ \cup \emptyset \\ \mathtt{num}^+ \cup \mathtt{nil}^- \cup \emptyset \\ \mathtt{num}^- \cup \mathtt{nil}^+ \cup \emptyset \\ \mathtt{num}^- \cup \mathtt{nil}^- \cup \emptyset\end{array}$$

Types for the Core Calculus VI

Types of Checked and Unchecked Primitives

$$
\begin{array}{lcl}
\mathit{TypeOf}(\mathbf{0}) & = & \forall \alpha. \\
 & & \mathtt{num}^{+} \cup \alpha \\
\mathit{TypeOf}(\mathbf{add1}) & = & \forall \alpha_1 \alpha_2 \varphi. \\
 & & ((\mathtt{num}^{\varphi} \cup \emptyset)\mathtt{->}^{+}(\mathtt{num}^{+} \cup \alpha_1)) \cup \alpha_2 \\
\mathit{TypeOf}(\mathbf{number}?) & = & \forall \alpha_1 \alpha_2 \alpha_3. \\
 & & (\alpha_1 \mathtt{->}^{+}(\mathtt{true}^{+} \cup \mathtt{false}^{+} \cup \alpha_2)) \cup \alpha_3 \\
 & & \\
\mathit{TypeOf}(\mathbf{CK\text{-}add1}) & = & \forall \alpha_1 \alpha_2 \alpha_3 \varphi. \\
 & & ((\mathtt{num}^{\varphi} \cup \alpha_3)\mathtt{->}^{+}(\mathtt{num}^{+} \cup \alpha_1)) \cup \alpha_2
\end{array}
$$

Typing Rules

$$(\textbf{const})\ \frac{\tau \prec \mathit{TypeOf}(c)}{A \vdash c : \tau} \qquad (\textbf{var})\ \frac{\tau \prec A(x)}{A \vdash x : \tau}$$

$$(\textbf{ap})\ \frac{A \vdash e_1 : (\tau_2 \text{->}^f \tau_1) \cup \emptyset \quad A \vdash e_2 : \tau_2}{A \vdash (\textbf{ap}\ e_1\ e_2) : \tau_1}$$

$$(\textbf{Cap})\ \frac{A \vdash e_1 : (\tau_2 \text{->}^f \tau_1) \cup \tau_3 \quad A \vdash e_2 : \tau_2}{A \vdash (\textbf{CHECK-ap}\ e_1\ e_2) : \tau_1}$$

$$(\textbf{lam})\ \frac{A, x : \tau_2 \vdash e : \tau_1}{A \vdash (\textbf{lambda}\ (x)\ e) : (\tau_2 \text{->}^+ \tau_1) \cup \tau_3}$$

$$(\textbf{let})\ \frac{A \vdash e_1 : \tau_1 \quad A, x : \forall \vec{\alpha}\vec{\varphi}\tau_1 \vdash e_2 : \tau_2 \quad \vec{\alpha}\vec{\varphi} = fv(\tau_1) \setminus fv(A)}{A \vdash (\textbf{let}\ ([x\ e_1])\ e_2) : \tau_2}$$

Type Soundness

- 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 $e \longrightarrow^* \textbf{error}$ or $e \longrightarrow^* v$ 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 ($\tilde{\nu}$ in the rules).
- Typing rules check absent variables for emptiness and chose checked or unchecked versions as appropriate.
- Example

$$
\begin{array}{lcl}
\textit{TypeOf}(\textbf{add1}) & = & \forall \alpha_1 \alpha_2 \varphi. \\
 & & ((\mathtt{num}^{\varphi} \cup \emptyset) \texttt{->}^{+} (\mathtt{num}^{+} \cup \alpha_1)) \cup \alpha_2 \\
\textit{SoftTypeOf}(\textbf{add1}) & = & \forall \alpha_1 \alpha_2 \bar{\alpha}_3 \varphi. \\
 & & ((\mathtt{num}^{\varphi} \cup \bar{\alpha}_3) \texttt{->}^{+} (\mathtt{num}^{+} \cup \alpha_1)) \cup \alpha_2
\end{array}
$$

Soft Typing Transformation Rules

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

$$(\textbf{var})\ \frac{\tau \prec A(x)}{A \vdash_s x \Rightarrow x : \tau}$$

$$(\textbf{ap})\ \frac{A \vdash_s e_1 \Rightarrow e_1' : (\tau_2 {-}{>}^f \tau_1) \cup \tilde{\tau}_3 \quad A \vdash_s e_2 \Rightarrow e_2' : \tau_2}{A \vdash_s (\textbf{ap}\ e_1\ e_2) \Rightarrow (\mathit{empty}\{\tilde{\tau}_3\} \rightarrow (\textbf{ap}\ e_1'\ e_2'), (\textbf{CHECK-ap}\ e_1'\ e_2')) : \tau_1}$$

$$(\textbf{Cap})\ \frac{A \vdash_s e_1 \Rightarrow e_1' : (\tau_2 {-}{>}^f \tau_1) \cup \tau_3 \quad A \vdash_s e_2 \Rightarrow e_2' : \tau_2}{A \vdash_s (\textbf{CHECK-ap}\ e_1\ e_2) \Rightarrow (\textbf{CHECK-ap}\ e_1'\ e_2') : \tau_1}$$

$$(\textbf{lam})\ \frac{A, x : \tau_2 \vdash_s e \Rightarrow e' : \tau_1}{A \vdash_s (\textbf{lambda}\ (x)\ e) \Rightarrow (\textbf{lambda}\ (x)\ e') : (\tau_2 {-}{>}^+ \tau_1) \cup \tau_3}$$

$$(\textbf{let})\ \frac{\begin{array}{c} A \vdash_s e_1 \Rightarrow e_1' : \tau_1 \quad A, x : \forall \vec{\alpha}\vec{\varphi}\tau_1 \vdash_s e_2 \Rightarrow e_2' : \tau_2 \\ \vec{\alpha}\vec{\varphi} = (\mathit{fv}(\tau_1) \setminus \mathit{fv}(A)) \setminus \mathit{AbsentVar} \end{array}}{A \vdash_s (\textbf{let}\ ([x\ e_1])\ e_2) \Rightarrow (\textbf{let}\ ([x\ e_1'])\ e_2') : \tau_2}$$

Related Work

- Robert Cartwright and Mike Fagan. Soft Typing. PLDI 1991.
- 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).
- Carsten K. Gomard. Partial Type Inference for Untyped Functional Programs. LFP 1990.