Points-to Analysis for JavaScript*

Dongseok Jang
Dept. of Computer Science
Korea Advanced Institute of Science & Technology
dsjang@kaist.ac.kr

Kwang-Moo Choe
Dept. of Computer Science
Korea Advanced Institute of Science & Technology
choe@kaist.ac.kr

ABSTRACT

JavaScript is widely used by web developers and the complexity of JavaScript programs has increased over the last year. Therefore, the need for program analysis for JavaScript is evident. Points-to analysis for JavaScript is to determine the set of objects to which a reference variable or an object property may point. Points-to analysis for JavaScript is a basis for further program analyses for JavaScript. It has a wide range of applications in code optimization and software engineering tools. However, points-to analysis for JavaScript has not yet been developed.

JavaScript has dynamic features such as the runtime modification of objects through addition of properties or updating of methods. We propose a points-to analysis for JavaScript which precisely handles the dynamic features of JavaScript. Our work is the first attempt to analyze the points-to behavior of JavaScript. We evaluate the analysis on a set of JavaScript programs. We also apply the analysis to a code optimization technique to show that the analysis can be practically useful.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms,Design,Experimentation,Languages

Keywords

JavaScript, points-to analysis, pointer analysis, program analysis, scripting language

*This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-02004-0.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

1. INTRODUCTION

JavaScript is a scripting language designed for client-side web scripting. There is now a standardized version, ECMAScript[6]. JavaScript has attracted more users and JavaScript programs have become lengthy and complex. Almost all web browsers support JavaScript. With help of the DOM[11] and Ajax[7], there are more and more sophisticated JavaScript programs used in popular web sites in these days. The web sites use JavaScript to implement important application logic rather than simple user interfaces.

A problem of JavaScript programs is slow execution speed. That is because JavaScript programs are usually executed by interpreters and JavaScript has many dynamic features which must be checked at runtime. The speed of JavaScript programs affects people's perception about the responsiveness of popular websites.

Performance improvement through the use of code optimization is an important method for making JavaScript a proper choice for building high quality software. Because a JavaScript statement executes many machine instructions, a little change of a JavaScript source code can bring about much improvement of the performance. Code optimization can be statically applied by using source level transformation. JavaScript compilers can also adopt code optimization to generate faster target code. Even JavaScript interpreters can utilize code optimization techniques at runtime.

Points-to analysis for JavaScript is essential for code optimization, but it has not yet been developed. Points-to analysis for JavaScript determines the set of objects to which a reference variable or an object property may point. Points-to analysis enables essential analyses for code optimization, such as side-effect analysis and def-use analysis.

In this paper, we present and evaluate a points-to analysis for JavaScript as a first step for further program analyses for JavaScript. Our analysis is based on Andersen's points-to analysis for C[2]. In Section 2, we discuss a motivating example of our research. Then, we define a restricted language to briefly describe points-to behavior of JavaScript in Section 3. We present a constraint-based, flow- and context-insensitive [1] points-to analysis for the restricted language in Section 4. In Section 5, we evaluate our analysis on a set of JavaScript programs. We also evaluate the impact of the analysis on a special case of partial-redundancy elimi-

[1]A flow-insensitive analysis does not take control-flow into account. A context-insensitive analysis does not distinguish between different invocations of a function

```
S0: var str = prompt();
S1: var a = new Object();  // o1
S2: a.x = new Object();    // o2
S3: a.y = new Object();    // o3
S4: a[str] = new Object(); // o4
S5: b = a.x;
```

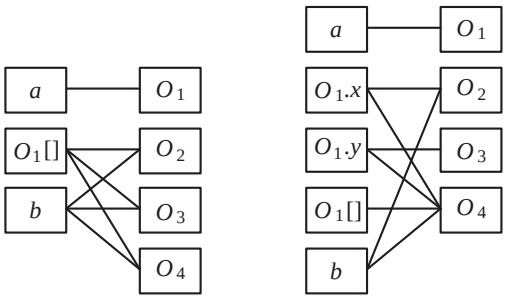

Figure 1: Example of JavaScript program and its points-to graphs. Top: Program code, Bottom left: Conventional graph, Bottom right: Graph with considering properties

nation[1] for the JavaScript programs. Section 6 discusses related work. Finally, Section 7 presents conclusions and future work.

2. MOTIVATION

In a sense, a JavaScript object is an associative array–a data structure that allows to dynamically associate arbitrary data values with arbitrary strings. An object property[2] can be accessed as an array element. The feature is represented in JavaScript syntax. For example, the JavaScript expression `object.property` is equivalent to `object["property"]`.

In Figure 1, the program shows that behavior. *S0* gets a string from a library function. Then, *S1* creates a new object o_1 with no properties. Here we name an object by its allocation site in a program. *S2* assigns o_2 to a non-existing property named `x` of o_1 referenced by `a`. Because it does not exist, the property is created on the fly and the value is assigned to the newly created property. *S3* does similar operations to the property `y` of o_1. *S4* assigns o_4 to a property whose name is given by the expression `str` via the [ ] operator. The expression `str` may evaluate to `"x"`,`"y"`, or a non-existing property name. When an object property is accessed by the [ ] operator, the actual property name may be statically unknown.

Conventional points-to analyses based on Andersen's analysis[2] treat elements of an array as an aggregate. If we naively adapt the conventional analyses to JavaScript, object properties are also treated as aggregates because object properties are only array elements indexed by strings in JavaScript. In practice, object properties are used for complex hierarchical data structures such as the DOM[11]. Therefore, the results of the conventional analyses would be inaccurate. For the program in Figure 1, Andersen's algorithm computes the points-to graph on the bottom left. In the points-to graph, all the properties of o_1 are aggregated as $o_1[\,]$. Consequently, the points-to set of `b` is $\{o_2, o_3, o_4\}$.

[2]Fields of an object are called "properties" of the object in JavaScript

If we maintain information for each property to increase the precision of the analysis, we obtain the points-to graph on the bottom right of Figure 1. In the graph, we maintain the name of each property of an object. For example, we use $o_1.x$ to represent the property `x` of the object o_1. For the added property via the [ ] operator, we use the aggregate $o_1[\,]$ in the same way of Andersen's analysis.

When an object property is updated via the [ ] operator, we may not know what property of the object is actually updated because the name of the changing property may not be statically determined. Any existing property of the object may be updated, or a new property may be created in the object. Because `str` may evaluate to `"x"` or `"y"` at *S4*, $o_1.x$ and $o_1.y$ point to o_4 in our points-to graph for the program. The node $o_1[\,]$ is for the case that `str` evaluate to a property name which cannot be statically determined. In our points-to graph, the points-to set of `b` is $\{o_2, o_4\}$. This is more accurate than the conventional approaches. Our points-to analysis increases accuracy by distinguishing each property separately while considering dynamic features of JavaScript.

3. SIMPLESCRIPT

For presentation brevity, we define SimpleScript, a restricted language of JavaScript. The most part of SimpleScript is based on Thiemann's work [16], but modified in some ways. We add the . operation and a unique global object to SimpleScript to expose significant points-to behaviors of JavaScript.

JavaScript is a weakly and dynamically typed object-based language. JavaScript has no classes but supports constructors and prototyping to share functionality of code. JavaScript provides the runtime modification of objects through addition of properties or updating of methods. A JavaScript object is just like an associative array– a data structure that allows to dynamically associate arbitrary data values with arbitrary strings(property names).

JavaScript has lexically scoped first-class functions which behave as functions or methods. When a function object is assigned to a property of an object, the function acts as a method if it is referenced by the property of the object and called. If a function is called as a method of an object, each reference to `this` is bound to the object in the function body. Otherwise, each reference to `this` resolves to the unique global object of JavaScript. A function can be used as a constructor when invoked through the `new` operator. The `new` operator creates a new object, and calls a constructor which binds `this` to the new object in the function body.

JavaScript has a unique global object. Whenever a variable is declared in the global scope or a value is assigned to an undeclared variable, the variable becomes a property of the global object.

The abstract syntax of SimpleScript is in Figure 2. SimpleScript comprises the following language constructs.

- `this` : When a function is called as a method of an object, `this` is a reference to the object receiving the method call in the function body. Otherwise, `this` is a reference to the global object.
- {} is an object literal which creates a new object with no properties.
- `function` $f(x_1, ..., x_m)\{s\}$ creates a new function object with the formal parameters $x_1, ..., x_m$ and the func-

Expressions

$e ::=$	`this`	self reference in method calls
$\mid$	x	variable
$\mid$	c	primitive data value
$\mid$	$\{\}$	object literal
$\mid$	`function` $x(x,...)\{s\}$	function expression
$\mid$	$e.x$	property reference by .
$\mid$	$e[e]$	property reference by [ ]
$\mid$	`new` $e(e,...)$	object creation
$\mid$	$e(e,...)$	function call
$\mid$	$e = e$	assignment
$\mid$	$p(e,...)$	primitive operators(add, etc.)

Statements

$s ::=$	`skip`	no operation
$\mid$	`var` x	variable declaration
$\mid$	e	expression statement
$\mid$	$s; s$	sequence
$\mid$	`if` (e) `then` $\{s\}$ `else` $\{s\}$	conditional
$\mid$	`while` (e) $\{s\}$	iteration
$\mid$	`return` e	function return

Figure 2: Syntax of SimpleScript

tion body s. The statements in the body refer to the enclosing function as f, but f is invalid outside of the function.

- $e.x$ is a property reference by the . operator. The expression e evaluates to an object and x is a property name. If the expression appears on the left side of an assignment expression and the property x is not in the object, the property is created at runtime.
- $e_1[e_2]$ is a property reference by the [ ] operator. The expression e_1 evaluates to an object. e_2 evaluates to a string which is used as a property name of the object. If e_2 evaluates to the string x, the meaning of the entire expression is equivalent to $e_1.x$.
- `new` $e_0(e_1,...)$ is an object creation expression. e_0 evaluates to a function object and the function is called with $e_1,...$ as its actual parameters. In the function body, `this` resolves to the newly created object for the object creation expression. If the function does not return an object, then the entire expression evaluates to the new object. Otherwise, the entire expression evaluates to the return value of the function.
- $e_0(e_1,...)$ is a function call. e_0 evaluates to a function object. The function is called with $e_1,...$ as its actual parameters. If the function is called as a method of an object, that is, e_0 is the form of $e_{01}.x$ or $e_{01}[e_{02}]$, `this` is bound to an object to which e_{01} evaluates in the function body. Otherwise, `this` is bound to the global object.
- $p(e_1,...)$ is for primitive operations such as arithmetic or logical operations. SimpleScript primitive operations return a primitive data value. [3]
- `var` x is a variable declaration. When a variable is declared in the global scope or a value is assigned to an undeclared variable, the variable becomes a property of the global object.

[3]In JavaScript, the logical AND(&&) and logical OR(||) operations may return an object.

For presentation brevity, some JavaScript features are restricted in SimpleScript. All identifiers are assumed to be syntactically different and a global variable is declared before it is used in a SimpleScript program. SimpleScript does not allow prototyping and the implicit conversion from a primitive data value to an object. Property deletion is meaningless to our flow-insensitive analysis and so is not included in our analysis. Those restricted features can be easily added to our analysis with trivial modification.

4. POINTS-TO ANALYSIS

In this section, we present a flow- and context-insensitive points-to analysis for JavaScript. We develop our analysis based on the set-constraint framework[8]. The analysis precisely models dynamic features of JavaScript, such as the runtime modification of objects through addition of properties or updating of methods. Our analysis maintains information for each property of an object to increase the precision of points-to analysis.

We present set constraints for the points-to analysis (Section 4.1). Then, we describe our set-constraints system designed in the form of constraint generation rules. Our analysis constructs set constraints for input programs by using the generation rules (Section 4.2). Next, by using a set of rules for solving, the analysis computes the least solution(or model) of the constraints (Section 4.3).

4.1 Set Constraints

A set constraint is of the form $\mathcal{X}_e \supseteq se$ where $\mathcal{X}_e$ is a set variable and se is a set expression. A set constraint $\mathcal{X}_e \supseteq se$ is read "at runtime, the expression e evaluates to an object in a set of objects(a points-to set) including those of se." We write $\mathcal{C}$ for a finite collection of set constraints.

The syntax and semantics of set expressions are in Figure 3. The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values (subsets of Val). In the definition of $\mathcal{I}$, some set expressions themselves impose restrictions on $\mathcal{I}$. If these restrictions are not met, then the interpretation of the expression is undefined. We call an interpretation $\mathcal{I}$ a *model*(a solution) of a conjunction of constraints $\mathcal{C}$ if, for each constraint $\mathcal{X} \supseteq se$ (set variable $\mathcal{X}$ and set expression se) in $\mathcal{C}$, $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. We write $lm(\mathcal{C})$ for the least model of $\mathcal{C}$. The semantics of set expressions naturally follows from their corresponding language constructs.

- $\mathcal{X}_e$: A set variable for objects to which e evaluates.
- $\mathcal{X}_{e:recv}$: If e is a property reference, *i.e.*, e is $e_1.x$ or $e_1[e_2]$ and $e()$ is executed, then `this` is bound to the value of e_1 in the called function body. When a function referenced by e is called, $\mathcal{X}_{e:recv}$ is used for passing the object receiving the method call to `this` in the function body.
- $\mathcal{X}_{o.x}$: A set variable for objects assigned to the object o's property x via the . operator. If the variable `l` refers to the object o, then the statement `l.x = r` let $\mathcal{X}_{o.x}$ contain $\mathcal{X}_r$
- $\mathcal{X}_{o:dot}$: A set variable for all objects assigned to a property of the object o via the . operator. If the variable `l` refers to the object o, the statement `l.`*property* `= r` let $\mathcal{X}_{o:dot}$ contain $\mathcal{X}_r$, regardless of *property*.

Syntax of set expressions

$e \in Expressions$	SimpleScript expression
$o \in \{o_l \mid l \text{ is an object allocation site}\}$	object name
$x \in Identifiers$	SimpleScript identifier
$se ::= \mathcal{X}_e$	set variable for SimpleScript expression
$\mid \mathcal{X}_{e:recv}$	set variable for binding `this` for function call
$\mid \mathcal{X}_{o.x}$	set variable for property x of object o
$\mid \mathcal{X}_{o:dot}$	set variable for all properties of o assigned by .
$\mid \mathcal{X}_{o:brk}$	set variable for all properties of o assigned by [ ]
$\mid \mathcal{X}_{o:this}$	set variable for `this` in body of function o
$\mid \mathcal{X}_{o:return}$	set variable for return value of function o
$\mid const$	set expression for primitive data values
$\mid o$	set expression for object name
$\mid WriteDot(\mathcal{X}_{e_1}, x, \mathcal{X}_{e_2})$	set expression for property write by .
$\mid WriteBrk(\mathcal{X}_{e_1}, \mathcal{X}_{e_2})$	set expression for property write by [ ]
$\mid ReadDot(\mathcal{X}_{e_1}, x)$	set expression for property read by .
$\mid ReadBrk(\mathcal{X}_{e_1})$	set expression for property read by [ ]
$\mid Call(\mathcal{X}_{e_0}, \mathcal{X}_{e_0:recv}, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n})$	set expression for function call

Semantics of set expressions

Val	$= \{o_l \mid l \text{ is an object allocation site}\} \cup \{const\}$
$DefOf$	$: Val \rightarrow Expressions$
$\mathcal{I}(\mathcal{X}_e)$	$\subseteq Val$
$\mathcal{I}(const)$	$= \{const\}$
$\mathcal{I}(o_l)$	$= \{o_l\}$
$\mathcal{I}(WriteDot(\mathcal{X}_{e_1}, x, \mathcal{X}_{e_2}))$	$= \{\}$ provided $o \in \mathcal{I}(\mathcal{X}_{e_1})$ implies that $\mathcal{I}(\mathcal{X}_{o.x}) \supseteq \mathcal{I}(\mathcal{X}_{e_2}) \wedge \mathcal{I}(\mathcal{X}_{o:dot}) \supseteq \mathcal{I}(\mathcal{X}_{e_2})$
$\mathcal{I}(WriteBrk(\mathcal{X}_{e_1}, \mathcal{X}_{e_2}))$	$= \{\}$ provided $o \in \mathcal{I}(\mathcal{X}_{e_1})$ implies that $\mathcal{I}(\mathcal{X}_{o:brk}) \supseteq \mathcal{I}(\mathcal{X}_{e_2})$
$\mathcal{I}(ReadDot(\mathcal{X}_{e_1}, x))$	$= \{v \mid v \in \mathcal{I}(\mathcal{X}_{o.x}) \cup \mathcal{I}(\mathcal{X}_{o:brk}), o \in \mathcal{I}(\mathcal{X}_{e_1})\}$
$\mathcal{I}(ReadBrk(\mathcal{X}_{e_1}))$	$= \{v \mid v \in \mathcal{I}(\mathcal{X}_{o:dot}) \cup \mathcal{I}(\mathcal{X}_{o:brk}), o \in \mathcal{I}(\mathcal{X}_{e_1})\}$
$\mathcal{I}(Call(\mathcal{X}_{e_0}, \mathcal{X}_p, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n}))$	$= \{v \mid v \in \mathcal{I}(\mathcal{X}_{o:return})\}$

provided $o \in \mathcal{I}(\mathcal{X}_{e_0}) \wedge DefOf(o) =$ `function` $x_0(x_1, ..., x_n)\{...\}$ implies that
$\mathcal{I}(\mathcal{X}_{o:this}) \supseteq \mathcal{I}(\mathcal{X}_p) \wedge \mathcal{I}(\mathcal{X}_{x_1}) \supseteq \mathcal{I}(\mathcal{X}_{e_1}) \wedge ... \wedge \mathcal{I}(\mathcal{X}_{x_n}) \supseteq \mathcal{I}(\mathcal{X}_{e_n})$

Figure 3: Set expressions for SimpleScript

- $\mathcal{X}_{o:brk}$: A set variable for all objects assigned to a property of the object o via the [ ] operator. If the variable `l` refers to the object o, the statement `l[`str`] = r` let $\mathcal{X}_{o:brk}$ contain $\mathcal{X}_r$ regardless of the value of str.
- $\mathcal{X}_{o:this}$: A set variable for objects to which `this` points in the function object o.
- $\mathcal{X}_{o:return}$: A set variable for objects which the function object o returns.
- $const$: A set expression for representing all the primitive data values (number, string, boolean, null). $const$ itself means the singleton set containing $const$.
- o_l : An object name created at the object allocation site l. l is uniquely determined by a location in a program code. o_l itself means the singleton set containing o_l.
- $WriteDot(\mathcal{X}_{e_1}, x, \mathcal{X}_{e_2})$: A set expression for representing the semantics of the expression $e_1.x = e_2$. For all o in $\mathcal{X}_{e_1}$, the set expression imposes some restrictions to collect values of $\mathcal{X}_{e_2}$ into $\mathcal{X}_{o.x}$ and $\mathcal{X}_{o:dot}$.
- $WriteBrk(\mathcal{X}_{e_1}, \mathcal{X}_{e_2})$: A set expression for representing the semantics of the expression $e_1[str] = e_2$. For all o in $\mathcal{X}_{e_1}$, the set expression imposes a restriction to collect values of $\mathcal{X}_{e_2}$ into $\mathcal{X}_{o:brk}$.
- $ReadDot(\mathcal{X}_e, x)$: A set expression for objects to which the expression $e.x$ evaluates. The set expression means a set which is the union of $\mathcal{X}_{o.x}$ and $\mathcal{X}_{o:brk}$ for all o in $\mathcal{X}_e$. $\mathcal{X}_{o:brk}$ is for the case that the property x of o is set by the [ ] operator.
- $ReadBrk(\mathcal{X}_e)$: A set expression for objects to which the expression $e[str]$ evaluates. The set expression means a set which is the union of $\mathcal{X}_{o:dot}$ and $\mathcal{X}_{o:brk}$ for all o in $\mathcal{X}_e$. When e evaluates the object o, $e[str]$ might evaluate to any property of o. Therefore, all properties of o should be considered.
- $Call(\mathcal{X}_{e_0}, \mathcal{X}_{e_0:recv}, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n})$: A set expression for the function call expression $e_0(e_1, ..., e_n)$. The first parameter of $Call$ is for a function object that is called. The second parameter is for objects receiving a method call when the function is called as a method. The other parameters are for the actual parameters of the function. $DefOf$ is a function that maps from an object name to the SimpleScript expression which creates the object. $DefOf$ is used for obtaining the function's signature. The set expression imposes restrictions for binding `this` and actual parameters of a function.

A solution of our analysis is defined to be the least model of a conjunction of constraints. A conjunction of constraints for a program guarantees the existence of its least solution because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable[8].

[Const] $$\triangleright c : \{\mathcal{X}_e \supseteq const\}$$

[PrimOp] $$\frac{\triangleright e_i : \mathcal{C}_i, i = 0, ..., n}{\triangleright p(e_1, ..., e_n) : \{\mathcal{X}_e \supseteq const\} \cup \mathcal{C}_1 \cup ... \cup \mathcal{C}_n}$$

[ObjLit] $$\triangleright \{\}_l : \{\mathcal{X}_e \supseteq o_l\}$$

[This] $$\frac{\texttt{o is the enclosing function object}}{\triangleright \texttt{this} : \{\mathcal{X}_e \supseteq \mathcal{X}_{o:this}, \mathcal{X}_{e:recv} \supseteq o_g\}}$$

[FuncExpr] $$\frac{\triangleright s_1 : \mathcal{C}_1}{\triangleright \texttt{function}\ x_0(x_1, ..., x_n)\{s_1\}_l : \{\mathcal{X}_e \supseteq o_l, \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_1}$$

[Write] $$\frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright x = e_1 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_1}, \mathcal{X}_x \supseteq \mathcal{X}_{e_1}, \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_1}$$

[PWrite(.)] $$\frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright e_1.x = e_2 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_2}, \mathcal{X}_e \supseteq WriteDot(\mathcal{X}_{e_1}, x, \mathcal{X}_{e_2}), \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[PWrite([ ])] $$\frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2 \quad \triangleright e_3 : \mathcal{C}_3}{\triangleright e_1[e_2] = e_3 : \{\mathcal{X}_e \supseteq \mathcal{X}_{e_3}, \mathcal{X}_e \supseteq WriteBrk(\mathcal{X}_{e_1}, \mathcal{X}_{e_3}), \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

[PRead(.)] $$\frac{\triangleright e_1 : \mathcal{C}_1}{\triangleright e_1.x : \{\mathcal{X}_e \supseteq ReadDot(\mathcal{X}_{e_1}, x), \mathcal{X}_{e:recv} \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[PRead([ ])] $$\frac{\triangleright e_1 : \mathcal{C}_1 \quad \triangleright e_2 : \mathcal{C}_2}{\triangleright e_1[e_2] : \{\mathcal{X}_e \supseteq ReadBrk(\mathcal{X}_{e_1}), \mathcal{X}_{e:recv} \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[New] $$\frac{\triangleright e_i : \mathcal{C}_i, i = 0, ..., n}{\triangleright \texttt{new}\ e_0(e_1, ..., e_n)_l : \{\mathcal{X}_e \supseteq o_l, \mathcal{X}_e \supseteq Call(\mathcal{X}_{e_0}, o_l, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n}), \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_0 \cup ... \cup \mathcal{C}_n}$$

[Call] $$\frac{\triangleright e_i : \mathcal{C}_i, i = 0, ..., n}{\triangleright\ e_0(e_1, ..., e_n) : \{\mathcal{X}_e \supseteq Call(\mathcal{X}_{e_0}, \mathcal{X}_{e_0:recv}, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n}), \mathcal{X}_{e:recv} \supseteq o_g\} \cup \mathcal{C}_0 \cup ... \cup \mathcal{C}_n}$$

[Return] $$\frac{\texttt{o is the enclosing function} \quad \triangleright e_1 : \mathcal{C}_1}{\triangleright \texttt{return}\ e_1 : \{\mathcal{X}_{o:return} \supseteq \mathcal{X}_{e_1}\} \cup \mathcal{C}_1}$$

[VarDecl] $$\frac{\texttt{the declaration is at top-level}}{\triangleright \texttt{var}\ x : \mathcal{X}_{o_g.x} \supseteq \mathcal{X}_x}$$

Figure 4: Constraint generation rules for SimpleScript

$$\frac{\mathcal{X}_{e_1} \supseteq \mathcal{X}_{e_2} \quad \mathcal{X}_{e_2} \supseteq o}{\mathcal{X}_{e_1} \supseteq o} \qquad \frac{\mathcal{X}_{e_1} \supseteq \mathcal{X}_{e_2} \quad \mathcal{X}_{e_2} \supseteq const}{\mathcal{X}_{e_1} \supseteq const}$$

$$\frac{\mathcal{X}_e \supseteq WriteDot(\mathcal{X}_{e_1}, x, \mathcal{X}_{e_2}) \quad \mathcal{X}_{e_1} \supseteq o}{\mathcal{X}_{o.x} \supseteq \mathcal{X}_{e_2} \quad \mathcal{X}_{o:dot} \supseteq \mathcal{X}_{e_2}}$$

$$\frac{\mathcal{X}_e \supseteq WriteBrk(\mathcal{X}_{e_1}, \mathcal{X}_{e_3}) \quad \mathcal{X}_{e_1} \supseteq o}{\mathcal{X}_{o:brk} \supseteq \mathcal{X}_{e_3}}$$

$$\frac{\mathcal{X}_e \supseteq ReadDot(\mathcal{X}_{e_1}, x) \quad \mathcal{X}_{e_1} \supseteq o}{\mathcal{X}_e \supseteq \mathcal{X}_{o.x} \quad \mathcal{X}_e \supseteq \mathcal{X}_{o:brk}}$$

$$\frac{\mathcal{X}_e \supseteq ReadBrk(\mathcal{X}_{e_1}) \quad \mathcal{X}_{e_1} \supseteq o}{\mathcal{X}_e \supseteq \mathcal{X}_{o:dot} \quad \mathcal{X}_e \supseteq \mathcal{X}_{o:brk}}$$

$$\frac{\mathcal{X}_e \supseteq Call(\mathcal{X}_{e_0}, \mathcal{X}_{e_0:recv}, \mathcal{X}_{e_1}, ..., \mathcal{X}_{e_n}) \quad \mathcal{X}_{e_0} \supseteq o \quad DefOf(o) = \texttt{function}\ x_0(x_1, ..., x_n)\{...\}}{\mathcal{X}_e \supseteq \mathcal{X}_{o:return} \quad \mathcal{X}_{o:this} \supseteq \mathcal{X}_{e_0:recv} \quad \mathcal{X}_{x_i} \supseteq \mathcal{X}_{e_i}, i = 1, ..., n}$$

Figure 5: Constraint solving rules

4.2 Constraint Generation

For every program expression, our analysis generates set constraints representing the semantics of the expression. Figure 4 shows the constraint generation rules for SimpleScript. For our analysis, every program expression e has a constraint: $\mathcal{X}_e \supseteq se$. The $\mathcal{X}_e$ is a set variable for the points-to set of the expression e. For each generation rule, the subscript e of $\mathcal{X}_e$ denotes the current expression to which the rule applies. Our analysis also generates set constraints representing the semantics of global variable declarations and function return statements. For the statements which are not described in Figure 4, our analysis simply collects the constraints for expressions contained in the statements.

The relation "$\triangleright e : \mathcal{C}$" is read "constraints $\mathcal{C}$ are generated from expression e". The global object is represented by o_g. The expressions which creates a new object are labeled l, an object allocation site. The labels are used as object names created by the expressions. Some rules generate constraints which contain compound set expressions such as $WriteDot$,$WriteBrk$, etc. The meanings of the constraints are explained in Section 4.1. The constraints are resolved into simpler constraints during constraint solving in Section 4.3. For the expression e which can evaluate to a function, the rules generate a constraint of the form $\mathcal{X}_{e:recv} \supseteq se$. When a function call $e()$ is found, $\mathcal{X}_{e:recv}$ is used for passing objects receiving a method call to `this` in the function body. $\mathcal{X}_{e:recv} \supseteq o_g$ is generated for an expression that is not a property reference because `this` is bound to the global object if a function is not called as a method.

The correctness of $\triangleright$ (i.e., the least model of constraints generated by $\triangleright$ for a program includes the actual values) is assumed. The set-based operational semantics can be defined based on the small-step operational semantics of Thiemann[16]. The set-based semantics can be used as an intermediate form to prove that our system is correct as those outlined in [9].

[PrimOp] is for primitive operations. In SimpleScript, primitive operations always return a primitive data value. [ObjLit] is straightforward. A new object is labeled an object allocation site l. [This] is simple too. For a function

object o, $\mathcal{X}_{o:this}$ is for `this` reference in the function body.

[Write] deals with an expression which assigns a value to a reference variable. Two constraints are generated by the expression. The first describes that the expression itself has the value of its right side, and the second describes that the expression assigns the value of its right side to the variable. [PWrite(.)] concerns an expression which assigns a value to an object property by the . operator. The rule generates the following constraints for the expression: 1) the assignment expression has the value of e_2. 2) for an object to which e_1 evaluates, the property x of the object has the value of e_2. [PWrite([ ])] is for an expression which assigns a value to an object property by the [ ] operator. The generated constraints describes the following: 1) the assignment expression itself has the value of e_2. 2) for an object to which e_1 evaluates, an unknown property of the object has the value of e_2. Moreover, any property of the object can be updated.

[PRead(.)] concerns a property reference by the . operator. The rule generates two constraints for the expression. The first constraint describes that the expression has the value of the property x of the object described by e_1. The second constraint is for binding `this` reference when a function referenced by the expression is called. When a function is referenced by an object o's property and is called, `this` is bound to the object o in the function body. The set variable $\mathcal{X}_{e:recv}$ is used for that binding. [PRead([ ])] is for a property reference by the [ ] operator. The rule generates two constraints for the expression. The first constraint describes that the expression has the value of any property of the object described by e_1. The second constraint is the same as the one for [PRead(.)].

[FuncExpr] is for a function expression. The function expression is labeled l, and evaluates to a newly created function object o_l. [Call] is for a function call. From the expression, the rule generates a constraint which contains a set expression $Call$. The meaning of the constraint is described in Section 4.1. [New] is for an object creation expression. A new object o_l is created and `this` is bound to the new object in the body of the function used as the constructor. If the function does not return an object value, the expression evaluates to the new object. Otherwise, the expression evaluates to the return value of the function. The constraints describes those behaviors.

[Return] is the rule for a return statement. [VarDecl] is for a global variable declaration. If a variable is declared in global scope, it becomes a property of the global object. $\mathcal{X}_{o_g.x}$ is for the property x of the global object.

4.3 Constraint Solving

This section presents constraint solving rules for our constraint systems. Set constraints containing compound set expressions are resolved into simpler constraints by using the rules in this section. Figure 5 shows the constraint solving rules for SimpleScript. Intuitively, the rules propagate values along all the possible data flow paths in a program.

The first two rules simply propagate values. The rule for $WriteDot$ introduces $\mathcal{X}_{o.x} \supseteq \mathcal{X}_{e_2}$ for the property x of o, and $\mathcal{X}_{o:dot} \supseteq \mathcal{X}_{e_2}$ to collect the object o's properties which are set by the . operator. The rule for $WriteBrk$ collects in $\mathcal{X}_{o:brk}$ the object o's properties which are set by the [ ] operator. The rule for $ReadDot$ introduces $\mathcal{X}_e \supseteq \mathcal{X}_{o.x}$ for the property x's values which are explicitly set by the . operator. $\mathcal{X}_e \supseteq \mathcal{X}_{o:brk}$ is added by the rule because the property x can be implicitly set by the [ ] operator. All properties of the object o consist of properties defined by the . operator ($\mathcal{X}_{o:dot}$) or the [ ] operator ($\mathcal{X}_{o:brk}$). The rule for $ReadBrk$ just collects all properties of the object o because the [ ] operator may return a value of any property of an object. The rule for $Call$ adds new constraints representing parameter bindings. $DefOf$ is used to obtain the function signature. The $DefOf$ information can be simultaneously collected in the constraint generation phase of Section 4.2.

The solution can be computed by the conventional iterative fixpoint method because the solution space is finite : object names given by object allocation sites. Correctness proof can be done by the fixpoint induction over the continous functions that are derived[4] from our constraint systems.

5. EXPERIMENTAL RESULTS

We implemented a prototype points-to analyzer for JavaScript. We used the JavaScript interpreter Rhino[15] to parse a JavaScript program. All experiments were conducted on a 3.0Ghz Intel Pentium 4 machine with 1Gb physical memory. We used SunSpider[14], one of the most popular JavaScript benchmarks. The benchmark programs are implemented in pure JavaScript without using the DOM scripting or other browser APIs. Table 1 shows the benchmark programs. The column "Object creation sites" reports the number of expressions which creates a new object. We exclude programs which heavily depend on a particular API like RegExp, and programs which do not have more than one object creation statement because they do not show significant points-to behavior. We also exclude programs with higher-order script because they are out of our scope.

5.1 Analysis Time and Object Read-Write Information

The third column of Table 1 shows the analysis time of our analysis. The reported times are the median values out of 5 runs. For all the programs, the analysis runs in less than 200 microseconds. The results show that our analysis can be done even at runtime for a small but important JavaScript program.

We measured object read-write information to estimate the preciseness of our analysis. We considered all *indirect access expressions*, expressions of the form $a.b$ or $a[e]$ in a program. For each such expression, the points-to set of a contains objects which may be read or written by the expression. Clients of object read-write information perform better with smaller points-to sets. Similar metrics have been used for a points-to analysis for C[10] or Java[12] to perform measurements to estimate the impact of the analysis.

The experimental results are summarized in Table 1. The column "Avg # of Objects for Indirect Access" is for the average number of accessed objects for indirect access expressions in a program. It is less than 2 in 10 out of the 13 programs. These results show that the analysis precisely models the points-to behavior of the programs, and that the points-to solution can be effectively used for applications of points-to analysis such as call graph construction.

5.2 Eliminating Redundant Property References

Program	Lines	Object Creation Sites	Analysis Time (ms)	Avg # of Objects for Indirect Access
cube	346	112	146	4.64
morph	35	5	31	1
raytrace	448	105	168	3.58
binary-trees	57	10	62	1.73
fannkuch	73	8	31	1
nbody	175	31	78	2.36
nsieve	45	9	31	1
aes	432	81	125	1.68
md5	301	45	68	1.14
sha1	231	41	73	1.25
spectral-norm	58	15	36	1.17
fasta	93	14	36	1.38
validate-input	95	13	46	1

Table 1: Characteristics of programs and analysis time of the analysis and average number of accessed objects for indirect access expressions

A property reference is a basic language construct in JavaScript. In JavaScript programs, a property reference is a frequently used operation, and syntactically similar property references are commonly found. Therefore, redundant property references are good targets for partial-redundancy elimination[1]. For partial redundancy elimination, points-to analysis is essential in detecting whether or not a property reference evaluates to the same value at different points of a program.

Figure 6 shows an example of eliminating redundant property references. In the left program, the same property reference `a.x` is used at *S6* and *S8*. If the value of `a.x` is not changed between *S6* and *S8*, we can optimize the program by storing the value of the property in the temporary variable `t`, and then use the value of `t` instead of reevaluating `a.x`. The value of `a.x` is changed in one of the the following cases: 1) when an assignment expression `a =` e or `a.x =` e is executed. 2) when the property `x` of an `a`'s alias is updated, that is, `d.x =` e is executed. 3) when a function call does the same operations described in the case 2. Based on the fact that `b` is not an alias of `a`, and `f` does not change the property `x` of its parameter, the original program can be transformed into the program in Figure 6 (right).

We transformed the benchmark programs by using the proposed technique with and without the points-to solution from our analysis. The transformation was done on source code level. The transformation without points-to information is done by suggesting that any two variables are aliases of each other and a function call affects values of all property references. We found redundant property references in 9 out of 13 programs. The execution time of each program was measured on Mozilla Firefox 2.0. The reported times are the averages of 50 runs.

Table 2 shows the experimental results. The columns "PR" and "AE" show the number of property references and the number of assignment expressions in each program, respectively. The number of assignment expressions is reported because our transformation technique introduces a new assignment expression to a program. The column "Time" is for the execution time. We use "-" to signify that a program is not transformed by the transformation technique. In some programs, the number of assignment expressions is increased without decreasing the number of property references because of eliminating loop invariant property references. The transformation technique simply tries to eliminate all redundant property references, even when a same property reference is used only twice. However, the simple method could make performance worse because introduced operations for temporary variables may cost more than the eliminated property references in some situations. For example, the performance becomes worse in the case of the transformed cube program without points-to information before the transformation. The performance improvement can be more significant if redundant property references are selectively eliminated by considering the trade-off between introducing assignment expressions and eliminating property references in various situations.

The transformation technique with points-to information eliminated more redundant property references and improved the performance of each program better than the technique without points-to information did. The results show that our points-to analysis can be useful for a practical application of the analysis.

```
S0: function f(p)
    { p.y = 0;}
S1: var a = {};
S2: var b = {};
S3: var c = {};
S4: var d = a;
S5: a.x = Math.rand();
S6: b.x = a.x * 0.5;
S7: f(a);
S8: c.x = a.x * 0.9;
```

```
S0: function f(p)
    { p.y = 0;}
S1: var a = {};
S2: var b = {};
S3: var c = {};
S4: var d = a;
S5: a.x = Math.rand();
S6: var t = a.x;
S7: b.x = t * 0.5;
S8: f(a);
S9: c.x = t * 0.9;
```

Figure 6: Example of eliminating redundant property references. Left:Original program, Right:Program transformed by eliminating redundant property references

Program	Original			Transformed without PTA			Transformed with PTA		
	PR	AE	Time (ms)	PR	AE	Time (ms)	PR	AE	Time (ms)
cube	346	178	868.6	323	192	995.1	221	224	781.3
raytrace	350	195	545.1	322	218	520.3	298	233	510.0
nbody	77	60	785.9	76	61	784.5	73	65	781.3
aes	169	168	400.1	169	169	393.8	148	191	364.3
md5	94	119	439.1	-	-	-	91	126	426.5
sha1	38	68	437.5	-	-	-	35	75	429.6
spectral-norm	16	27	446.9	14	28	435.8	12	30	431.2
fasta	12	34	856.4	-	-	-	12	35	837.6
validate-input	11	34	624.8	-	-	-	11	35	595.5

Table 2: Characteristics of original programs and programs transformed by eliminating redundant property references. "PR" is for the number of property references and "AE" is for the number of assignment expressions

6. RELATED WORK

Program analysis for C may be a starting point of program analysis for JavaScript. There are various points-to analyses for C with different tradeoffs between cost and precision. A relatively precise and efficient points-to analysis is Andersen's analysis for C[2]. Andersen's analysis is more precise than other works of Steensgaard[13] and Das[5] even though it is slower than the others. In addition, Andersen's analysis has been a starting point for other points-to analyses. Therefore, Andersen's approach is a reasonable basis for our analysis.

Because JavaScript is an object-based language, our work is clearly related to points-to analysis for other object-oriented languages. A points-to analysis for Java[12] is also based on Andersen's analysis. In [12], annotated constraints are used to track object properties separately. The approach cannot be directly adapted to JavaScript because of the runtime modification of objects. The potential impact of the approach are extensively measured by using object read-write information, call graph construction, and synchronization removal and stack allocation in [12].

The type systems for JavaScript[16, 3] focus on helping programmers debug and maintain JavaScript programs. They consider the runtime modification of objects. Thiemann[16] specifies a formal semantics for JavaScript and a type system. Thiemann's type system models the automatic type conversions of JavaScript to detect runtime errors such as accessing a property of the `null` object.

The type systems can also be used to execute JavaScript programs faster. While executing a JavaScript program, it takes much time for interpreters to do runtime type checking. With a static type system, a JavaScript interpreter can execute a program faster by avoiding runtime type checking for statically type checked variables.

7. CONCLUSIONS AND FUTURE WORK

We present a points-to analysis for JavaScript based on Andersen's points-to analysis for C[2]. We implement the analysis by using a constraint-based approach. Conventional points-to analyses treat elements of an array in aggregate. However, the conventional approaches would be inaccurate for JavaScript because JavaScript objects are also associative arrays. To distinguish between different properties of an object, our analysis carefully deals with an JavaScript object depending on whether the object is used as an array or not. We evaluate our analysis on JavaScript programs. We also apply our analysis to optimize the programs by eliminating redundant property references. Our results demonstrate that the analysis can be practically useful.

For future work we want to evaluate the impact of our analysis on a larget set of extensive JavaScript programs. We would need to model the points-to behavior of complicated hierarchical object structures such the DOM or browser-specific objects that are used in practical JavaScript programs. We would also like to develop various client analyses of points-to analysis to evaluate the effectiveness our analysis. Especially, we would like to develop general code optimization techniques for JavaScript, and evaluate the impact of the techniques on practical JavaScript programs.

8. ACKNOWLEDGMENTS

We are grateful to Dachuan Yu, Peter Thiemann, and Florian Loitsch for sharing their valuable ideas on this work.

9. REFERENCES

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2006.

[2] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[3] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. *19th European Conference on Object-Oriented Programming*, pages 428–453, 2005.

[4] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. *Lecture Notes in Computer Science*, 939:293–308, 1995.

[5] M. Das. Unification-based pointer analysis with directional assignments. *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, 2000.

[6] ECMA International. ECMAScript language specification. Stardard ECMA-262, 3rd Edition,, Dec 1999. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[7] J. Garrett. Ajax: A new approach to web applications, 2005.

[8] N. Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, 1992.

[9] N. Heintze. Set-based analysis of ML programs. *ACM SIGPLAN Lisp Pointers*, 7(3):306–317, 1994.

[10] M. Hind and A. Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, 25(5):113–123, 2000.

[11] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (DOM) level 3 core specification (W3C recommendation), 2004. http://www.w3.org/TR/DOM-Level-3-Core.

[12] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55, 2001.

[13] B. Steensgaard. Points-to analysis in almost linear time. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[14] The Webkit Open Source Project. Sunspider JavaScript benchmark, 2007. http://webkit.org/perf/sunspider-0.9/sunspider.html/.

[15] The Mozilla Organization. Rhino : JavaScript for Java, 2004. http://www.mozilla.org/rhino/.

[16] P. Thiemann. Towards a type system for analyzing JavaScript programs. *European Symposium On Programming*, pages 408–422, 2005.