JavaScript におけるプログラム変換の効果

田 村 知 博† 中 野 圭 介††
鵜 川 始 陽††† 岩 崎 英 哉†††

従来 JavaScript の実行効率を向上させるための技術としては，実行時情報を利用して最適化を行うことが重視されてきた．しかし，JavaScript をあらかじめコンパイルして実行する場合には，時間のかかる解析をともなう静的な最適化も有効であると考えられる．本稿では，Lambda Lifting と呼ばれるプログラム変換手法を JavaScript へ適用して既存処理系において実験を行い，効果の有無を議論する．実験結果から，Lambda Lifting を実行時情報を用いた最適化を行う処理系と併用することで，実行効率の向上が期待できると確認できた．

Effects of program transformation based on Lambda Lifting in JavaScript

Tomohiro TAMURA,† Keisuke NAKANO,††
Tomoharu UGAWA††† and Hideya IWASAKI†††

Dynamic optimizations using runtime information are standard techniques in existing JavaScript implementations. However, static program analysis can be effective when the programmer compiles JavaScript programs before running. In this report, we apply a program transformation called Lambda Lifting to JavaScript and discuss whether it is effective or not based on the experimental results. We found that Lambda Lifting increases execution speed of JavaScript programs that run on existing implementations with optimization based on runtime information.

1. はじめに

JavaScript[6)] は，主に Web アプリケーションにおいて，サーバとの通信や表示の切替えなどクライアント側で行う処理の記述に用いられる．そのため JavaScript 処理系は主に Web ブラウザに組み込まれ動作する．一方で，データベース へのアクセスなどサーバ側で行う処理の記述には Perl や PHP，Java など，JavaScript 以外の言語が用いられることが多い．

しかし，クライアントとサーバの両方を同じ言語で記述できれば，異なる言語を習得するための学習コストが減り，また，言語間の意味の違いに悩まずにすむため，Web アプリケーションの開発効率が向上すると考えられる．そのためサーバ側の処理も JavaScript で記述できることが望ましい．

サーバ側の処理を記述できる既存の JavaScript 処理系には Rhino[11)] と Node.js[9)] がある．Rhino は Java で記述されており，JavaScript プログラムを Java のクラスファイルへ変換するコンパイラと，クラスファイルを生成せずに実行するインタプリタを備える．Rhino はサーバ側で動作することを前提とした処理系であるが，Web ブラウザ上で動作する処理系に比べて遅いという問題点がある．Node.js は C++ で記述され，Web ブラウザ Google Chrome の JavaScript 処理系である V8 を流用し，サーバ側処理の記述に必要なライブラリを用意したフレームワークである．V8 は Web ブラウザ上で動作することを前提とした処理系であり，ユーザを待たせないようにするため，実行前にプログラム全体を対象とした解析を行わず，実行時情報に基づき部分的な最適化を行う．そのため，たとえば静的なプログラム変換のようなプログラム全体を

† 電気通信大学 大学院電気通信学研究科 情報工学専攻
Department of Computer Science, Graduate School of Electro-Communications, The University of Electro-Communications
†† 電気通信大学 先端領域教育研究センター
Center for Frontier Science and Engineering, The University of Electro-Communications
††† 電気通信大学 大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications

対象とした最適化は行わない．

こうした背景から，我々はサーバ側で動作することを前提とした JavaScript 処理系の開発を計画した．この処理系は，静的解析に基づく最適化を行うコンパイラと，実行時情報に基づく最適化を行う仮想マシンから構成する．しかし，JavaScript において静的解析による最適化と実行時情報に基づく最適化との併用が実行効率にどのような影響を与えるかについては自明でない．

そこで本稿では，静的なプログラム変換手法である Lambda Lifting[5] を JavaScript に応用し，小さなマイクロベンチマークプログラムおよび一般的なベンチマークに対する実験結果から，JavaScript に対するプログラム変換の効果について述べることを目的とする．

2. 既存処理系における最適化手法

Web ブラウザ上で動作する既存の主な JavaScript 処理系には，Google Chrome[8] の V8，Safari[12] の JavaScriptCore，Firefox[7] の SpiderMonkey，Opera[10] の Carakan などがある．本節ではこれらの処理系で主に用いられている最適化について述べる．

2.1 プロパティアクセスの高速化

JavaScript はプロトタイプベースのオブジェクト指向言語であり，オブジェクトに対して自由にプロパティの挿入・削除が行えるため，オブジェクトの構造が動的に変わりうる．ゆえに，オブジェクトのプロパティへアクセスする際には，アクセスのたびに探索を行ってプロパティの実体を見つける必要がある．オブジェクトへのプロパティアクセスは頻繁に起こるため，この探索処理を効率的に行うことが重要である．

これを解決するための方法として，プロトタイプベースのオブジェクト指向言語 Self での研究成果[2] が知られており，既存処理系においてもこれを踏襲している．この手法では，初期化を終えたオブジェクトに対するプロパティの追加・削除が頻繁に起こることはない，という仮定に基づき，実行時に，同じ構造を持つオブジェクトをグループ化する．グループ化はオブジェクトへのプロパティの追加・削除の順序に基づいて行う．すなわち，同じ初期化関数を用いて生成したオブジェクトのように，同じ名前のプロパティを同じ順序で追加したオブジェクトは，同じグループに属すようにする．このようにして分類したグループは，仮想的なクラスとみなすことができ，クラスベースのオブジェクト指向言語におけるインラインキャッシュ[4] という手法が適用できる．

インラインキャッシュは，プログラム中でプロパティ探索を行う位置ごとに，前回探索を行ったオブジェクトのクラスとプロパティの実体の位置をキャッシュしておく手法である．あるオブジェクトへのプロパティアクセスが必要になった時，オブジェクトのクラスが，そのコード位置での前回のアクセスと同じクラスであれば，探索をせずに実体を見つけられ，プロパティアクセスにかかる時間を短くできる．

2.2 効率の良いコードの生成

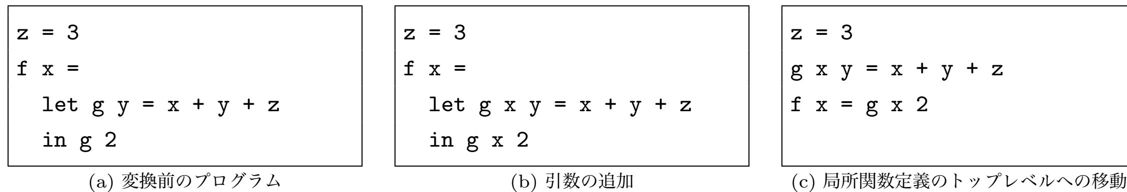
JavaScript は弱い動的型付けの言語であるため，変数に任意の型のデータを束縛できる．そのため変数に束縛されたデータの型を実行時に調べ，適切な処理を選ぶ必要がある．例えば `x+y` という式における `x` と `y` の型について，数値と数値であれば加算処理を，文字列と文字列であれば連結処理を行わなければならない．また，暗黙の型変換により，数値と真偽値，文字列と配列のような組み合わせが可能であるため，適切な処理を選ぶまでに調べなければならない組み合わせが多く，効率上の問題となる．

既存処理系では，JavaScript の構文木を解釈実行するのではなく，いったんバイトコードやネイティブコードへ変換して実行するが，このとき，数値同士や文字列同士などのよく現れる組み合わせを先に調べるコードを生成する．頻度が高いと思われる組み合わせを先に調べることで，実行速度の低下を防いでいる．

このコード生成をさらに最適化する手法として，実行時情報に基いた Type Specialization[1] がある．ある位置のコードを繰り返し実行する場合には同じ組み合わせが現れやすいという経験則に基づいて，型を調べる順序を変更したコードを動的に生成し，元のコードを置き換える．このことにより，以降，型を調べ適切な処理を選ぶのにかかる時間を短くできる．

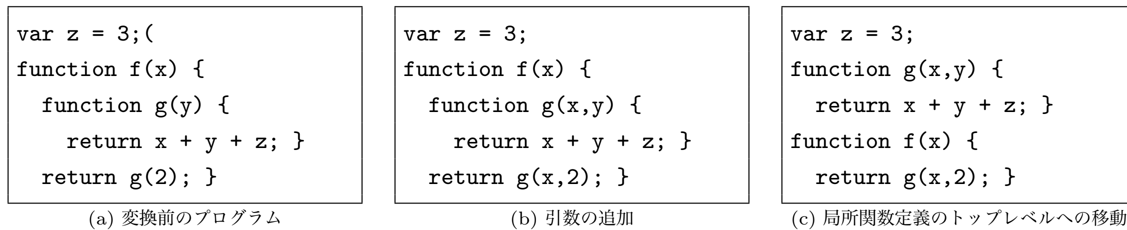
3. 本稿におけるプログラム変換

既存処理系は，前節で述べた実行時情報に基づく最適化の他，ランタイムシステムの効率化によって実行速度の向上を図っている．しかしながらこれらは静的なプログラム変換を行うものではない．そこで本稿では，JavaScript に対する静的なプログラム変換を適用して実験を行い，実行時情報に基づく最適化を行う処理系とプログラム変換を併用した場合における効果の有無を議論する．ここで静的な変換は JavaScript から JavaScript へのソースコード変換によって行い，実行は既存処理系の上で行う．プログラム変換手法としては Lambda Lifting[5] を応用し，局所関数定義をトップレベルへ移動する．



(a) 変換前のプログラム　(b) 引数の追加　(c) 局所関数定義のトップレベルへの移動

図 1　Haskell における Lambda Lifting の例
Fig. 1　Lambda Lifting in Haskell



(a) 変換前のプログラム　(b) 引数の追加　(c) 局所関数定義のトップレベルへの移動

図 2　JavaScript における Lambda Lifting の例
Fig. 2　Lambda Lifting in JavaScript

3.1 Lambda Lifting

本節では，Lambda Lifting の概要を述べる．Lambda Lifting は関数型言語における代表的なプログラム変換の 1 つである．Lambda Lifting では，局所関数内に現れる自由変数が束縛変数になるよう定義を書き換え，定義をトップレベルへ移動する．

定義の書き換えは以下の手順で行う．まず，局所関数定義内にあった自由変数を明示的に受け取れるよう，仮引数を増やす．次に，その局所関数の使用箇所において，追加した引数を明示的に与える．これらの書き換えののちに定義をトップレベルへ移動する．

図 1 に，Haskell における変換の例を示す．変換前のプログラムが図 1 (a) である．関数 `f` 内にある局所関数 `g` は，`x` と `z` を自由変数に持つ．大域変数である `z` は，`g` をトップレベルへ移動してもスコープを外れないため，図 1 (b) のように `g` の定義および `g` の呼び出し箇所に対して引数 `x` を追加する．最後に，図 1 (c) に示すように，スコープをはずれた自由変数を持たなくなった局所関数 `g` をトップレベルへ移動する．図 2 に，図 1 と同等の JavaScript における変換の例を示す．

3.2 メモリ管理コストに与える影響

本節では，JavaScript に対する Lambda Lifting がメモリ管理に与える影響について述べる．

JavaScript において関数定義は関数オブジェクトの生成を意味する．たとえば，以下の関数定義

```
function f(x) { return x; }
```

は，以下のコードと同等である．

```
var f = new Function("x","return x");
```

ただし，局所関数定義に対する関数オブジェクトの生成は，定義位置によらず，外側の関数の先頭で起こる．すなわち，ある関数が呼び出された際は毎回，関数本体の実行に先立って，その関数内にある局所関数定義に対応する関数オブジェクトを生成する．

Lambda Lifting により，局所関数定義をトップレベルへ移動すると，この関数に対する関数オブジェクトはプログラム実行開始時に 1 つ生成されるだけになる．このことにより関数オブジェクトの割当てコストを節約でき，またその結果としてごみ集め回数を減らせることで，実行時間の短縮が期待できる．

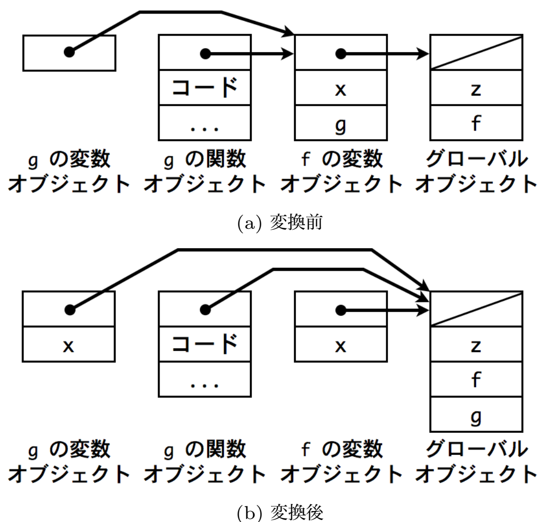
3.3 名前解決コストに与える影響

本節では，JavaScript に対する Lambda Lifting が名前解決コストに与える影響について述べる．

JavaScript において名前解決はオブジェクトのプロパティ探索で説明される．このことを以下のコードを用いて説明する．

```
var z;
function f(x) {
  function g() { return x+z; }
}
```

局所関数 `g` の関数オブジェクトとその環境を図示したのが図 3 (a) である．大域環境には，グローバル変数をプロパティとして保持するオブジェクトがあり，これをグローバルオブジェクトと呼ぶ．この例では，グローバルオブジェクトは `z` と `f` をプロパティとして持つ．関数の呼び出し時には，(関数名を含む）局所変数をプロパティとして持つオブジェクトが生成される．これを変数オブジェクトと呼ぶ．この例では，`f`



(a) 変換前

(b) 変換後

図 3 局所関数 g が持つ環境
Fig. 3 environment: local function g

の変数オブジェクトは x と g をプロパティとして持ち，g の変数オブジェクトはプロパティを持たない．f の実行時に生成される局所関数 g の関数オブジェクトは，環境として f の変数オブジェクトへの参照を持つ．また f の変数オブジェクトは，グローバルオブジェクトへの参照を持つ．この連鎖をスコープチェインと呼ぶ．g の実行時には，z の実体を取得するため，スコープチェインをたどる．すなわち，g と f の変数オブジェクトを調べたのちにグローバルオブジェクトを探索し，実体を見つける．

g に対して Lambda Lifting を行いトップレベルに移動すると，自由変数がスコープチェイン上のどこで見つかるかが変化する．変換後の環境を図示したのが図 3 (b) である．g へ明示的に与えるように変更した自由変数 x は，g の変数オブジェクトで見つかるようになる．また，g の変数オブジェクトがグローバルオブジェクトを直接参照するため，大域変数 z へのアクセスは f の変数オブジェクトを経由する必要がない．このことにより，変数の名前解決に要する時間を短縮できる．

3.4 JavaScript における変更点

Lambda Lifting を JavaScript へ適用するに当たり，効率上の理由から変更を加えた点がある．本節ではこの変更について述べる．

Lambda Lifting ではすべての局所関数をトップレベルへ移動するが，本稿のプログラム変換では局所関数定義を残す場合がある．

自由変数を持たない局所関数定義は仮引数を増やす

```
function f(x) {
  function g(y) { return x+y; }
  return g; }
}
```

(a) 局所関数オブジェクトのリターン

```
function g(x,y) { return x+y; }
function f(x) {
  return ( function(y) { g(x,y); } )
}
```

(b) 無名関数オブジェクトの生成と定義の移動

図 4 局所関数オブジェクトの関数外へのエスケープ
Fig. 4 a local function object escape from function

必要がないため，常にトップレベルへ移動する．一方，自由変数を持つ局所関数定義の場合，対応する関数オブジェクトの生存期間が，定義の存在する関数の呼び出し期間よりも長くなる場合には，トップレベルへ移動しない．具体的には，局所関数定義に対応する関数オブジェクトが戻り値のデータ構造に含まれる場合や，大域変数へ代入される場合を指す．

図 4 (a) に局所関数をトップレベルへ移動しない例を示す．自由変数を持つ局所関数 g の関数オブジェクトは f の戻り値に含まれている．Lambda Lifting のアルゴリズムで g をトップレベルへ移動するためには，g の定義へ仮引数を追加し，g の使用箇所へ実引数を追加する必要がある．もともと Lambda Lifting が対象としていた関数型言語には部分適用の機能があり，単純に g の使用箇所へ実引数を追加しさえすれば，引数が足りているかどうかを気にする必要はなく，必要に応じてクロージャが生成された．しかし JavaScript ではすべての関数が可変個引数であり，部分適用はなく定義の仮引数個数とそぐわない関数呼び出しであっても動作する．Lambda Lifting と同様にクロージャを生成するようにし，g をトップレベルへ移動すると図 4 (b) のようになる．戻り値には，無名関数を用いて生成した x への参照を持つ関数オブジェクトを渡している．このように変換した場合には，3.2 節で述べたメモリ管理コストの削減が見こめないことに加え，余計な関数呼び出しが一段挟まることによってオーバヘッドが生じる．

この例のように，戻り値に含まれるなどの理由により，関数 f の実行が終わったあとにも局所関数定義の関数オブジェクト g が生存することを，g が f からエスケープする，と呼ぶ．自由変数を持つ局所関数オブジェクトがエスケープする場合には，定義を移動

```
function f(x) {
  function g() { x = 1; }
  g();
  return x;
}
```

(a) 局所関数における自由変数への副作用

```
function g(x) { x = 1; }
function f(x) {
  g(x);
  return x;
}
```

(b) 定義の移動によるプログラムの意味の変化

図 5 自由変数への副作用
Fig. 5 side effect on free variable

するために無名関数オブジェクトを生成する必要があり，関数オブジェクトの生成を節約することができず，実行効率の向上が期待できない．

ゆえに本変換では，局所関数オブジェクトに対してエスケープ解析[3)]を行い，ある局所関数オブジェクトがいかなる実行経路においてもエスケープしない場合にのみ，その局所関数定義をトップレベルへ移動する．ただし関数間解析は行わず，任意の関数への引数として渡された局所関数オブジェクトは，すべてエスケープしたものとして扱う．これは，組み込みの関数であっても関数の定義が動的に変わりうるため，静的な解析が難しいことが理由である．

3.5 対象とするプログラム

JavaScript の持つ言語機構により，いくつかのケースでは本変換を施すことによってプログラムの意味が変わる．以下では，これらのケースについて議論する．

3.5.1 自由変数への副作用

本変換は，局所関数内において自由変数への副作用がある場合には正しく動作しない．たとえば図 5 (a) のプログラムは，局所関数 `g` 内で局所変数 `x` に対して代入を行っている．`g` はエスケープしていないため，トップレベルへ移動できると判断し変換を施すと図 5 (b) のようになる．図 5 (b) のプログラムでは `g` の呼び出しによる `x` への副作用が `f` へ波及せず，図 5 (a) と異なる結果を生む．そのため本変換では自由変数に対する副作用を含む局所関数をトップレベルに移動しない．

3.5.2 同値性判定

関数オブジェクト同士の同値性判定（`===`）では，ポインタ同値性によって判断を行う．関数呼び出しごとに異なるオブジェクトを生成していたプログラムは，本変換によって，同一のオブジェクトを使うようになるため，同値性判定の結果が変わる．エスケープした（自由変数を持たない）局所関数オブジェクトに対し同値性判定が行われるかどうか判断するためには大域的な解析が必要となるが，これは困難である．そのため本変換では局所関数オブジェクトに対する同値性判定が行われる場合，トップレベルに移動しない．

```
var x = 1;
function f(obj) {
  function g(obj) {
    with(obj) { return x; }
  }
  g(obj);
}
```

図 6 with 文を用いたプログラム
Fig. 6 a program using “with” statement

```
var y=0;
function f() {
  eval("var y=1;");
  function g() { return y; }
  return g();
}
```

図 7 eval 関数を用いたプログラム
Fig. 7 a program using “eval” function

3.5.3 `with` 文

JavaScript では，`with` 文を用いて動的に変数のスコープを変えることができる．図 6 のプログラムを用いて説明する．関数 `f` は引数を 1 つ受け取り，それを `with` 文に渡している．この `with` 文のブロック内では，スコープチェインの先頭に `obj` が追加された上で名前解決が起こる．つまり，`x` はまず `obj` の持つプロパティに存在するかどうかが調べられる．そのため，図 6 のような `with` 文を用いたプログラムに対し本変換を施した場合にはプログラムの意味が変わりうる．そのため本変換では `with` 文を用いたプログラムを対象としない．

ただし，JavaScript の次期仕様である ECMA Script 第 5 版においては，`with` 文が使用できなくなる strict モードが導入されるため，strict モードのプログラムを扱う場合には `with` 文に対する考慮は必要ない．

3.5.4 `eval` 関数

JavaScript では，`eval` 関数を用いて任意の文字列

```
function f() {
  function local_f() {}
  return local_f; }
for (var i; i < 1000000; i++) { f(); }
```

(a) 変換前

```
function moved_local_f() {}
function f() {
  var local_f = moved_local_f;
  return local_f; }
for (var i; i < 1000000; i++) { f(); }
```

(b) 変換後

図 8 メモリ管理コストに対するマイクロベンチマーク
Fig. 8 micro benchmark: memory management

を JavaScript プログラムとして実行することができる．図 7 のプログラムを用いて説明する．関数 `f` では構文解析の時点では局所変数宣言を持たないが，実行時，`eval` 関数によって局所変数 y が追加される．そのため局所関数 `g` 内の `y` は `eval` 関数によって追加された `y` を参照する．このケースにおいて本変換を適用するためには，`eval` に与えられた引数を解析する必要があるが，これは一般には困難である．また，ECMA Script 第 5 版における strict モードでは `eval` による局所変数の追加が制限されるが，代入文などのプログラムは実行できるため，strict モードにおいても解析は難しい．そのため本変換では `eval` 関数を用いたプログラムを対象としない．

4. 実　　験

JavaScript における Lambda Lifting の効果を計測するため実験を行った．実験環境は Intel Core i7 2.66 GHz，主記憶 8GB，Mac OS X 10.6.8 である．変換は Haskell で記述した変換器によって自動的に行った．

実験は，小さなマイクロベンチマークプログラムと，一般的に用いられているベンチマークプログラムを対象に行った．マイクロベンチマークを用いた実験では，3.2 節および 3.3 節で述べた影響が，既存の処理系において有効であるかどうかをそれぞれ確認した．一般的なベンチマークプログラムを用いた実験では，大きなプログラムにおいて効果が出るかどうかを確認した．

使用した処理系は node.js（V8），jsc（JavaScript-Core），SpiderMonkey，rhino の四種類である．

4.1 マイクロベンチマーク：メモリ管理

図 8 に実験に用いたプログラムを載せる．図 8 (a) が変換前，図 8 (b) が変換後のプログラムである．関数 `f` は局所関数 `local_f` を定義し（すなわち `local_f` に対応する関数オブジェクトを生成して），それを戻り値として返す．プログラムでは f を 100 万回呼び出す．変換前のプログラムでは `local_f` に対応する関数オブジェクトが 100 万個生成され，変換後のプログラムでは 1 個だけ生成される．

表 1 メモリ管理コストの比較
Table 1 comparing memory management cost

処理系	実行時間（秒）		実行時間の比
	変換前	変換後	(後/前)
node.js	0.226	0.066	0.291
jsc	0.333	0.048	0.145
SpiderMonkey	0.277	0.131	0.652
rhino	1.250	0.941	0.752

実験結果を表 1 に載せる．どの処理系においても，実行時間が有意に減少していることが見てとれる．また，Node.js では 9 回から 1 回，rhino では 22 回から 1 回と，GC 回数が減少していた．3.2 節で述べたとおり，実行時間の減少はメモリ管理コストの減少によるものと考えられる．実験結果から，実行時情報を用いた最適化を行う処理系と，静的なプログラム変換を組み合わせることにより，実行効率が向上する場合があることを確認できた．

4.2 マイクロベンチマーク：名前解決

図 9 に実験に用いたプログラムを載せる．図 9 (a) が変換前，図 9 (b) が変換後のプログラムである．関数 `f` の局所関数 `local_f` は大域変数 `global_x` を参照し，その値を返す．変換前のプログラムでは `global_x` にアクセスするため `f` の変数オブジェクトを経由する必要があるが，変換後のプログラムでは必要ない．

表 2 名前解決コストの比較
Table 2 comparing name resolution cost

処理系	実行時間（秒）		実行時間の比
	変換前	変換後	(後/前)
node.js	0.151	0.154	1.019
jsc	0.666	0.656	0.985
SpiderMonkey	1.431	1.441	1.006
rhino	3.766	1.778	0.472

実験結果を 表 2 に載せる．実行時間の有意な減少が見られたのは rhino のみで，他の処理系では，誤差の範囲でばらつきがあるものの，変換の前後で変化が見られなかった．このことは，node.js，jsc，Spider-Monkey の 3 つが，2.1 節で述べた仮想的なクラスの割当ておよびインラインキャッシュによってプロパティアクセスを高速化しているためと考えられる．この種

```
var global_x;
function f() {
  function local_f() {
     return global_x; }
  for (var i=0; i<10000000; i++) {
    local_f(); } }
f();
```

(a) 変換前

```
var global_x;
function moved_local_f() {
  return global_x; }
function f() {
  var local_f = moved_local_f;
  for (var i=0; i<10000000; i++) {
    local_f(); } }
f();
```

(b) 変換後

図 9　名前解決コストに対するマイクロベンチマーク
Fig. 9　micro benchmark: name resolusion

の最適化を行っていないであろう rhino については効果が現れたため，実行時情報を用いた最適化の効果を，静的なプログラム変換によっても得ることが可能であると示せた．すなわち，実行時情報を用いた最適化を静的なプログラム変換によって肩代わりすれば，実行時最適化のための負荷を減らせ，実行効率の向上が期待できる．

4.3　一般的なベンチマーク

処理系の性能を計る目的で一般的に用いられているベンチマークプログラムに対して本手法を施し，実行時間を測定した．対象のプログラムは Dromaeo，SunSpider，V8 の三種類，計 38 個であり，そのうち局所関数定義のトップレベルへの移動に成功した 7 個のプログラムに対して実験を行った．

図 10 は実行時間の比（変換後 / 変換前）をグラフにしたものである．どのプログラムも，誤差によるばらつきはあるものの変換の前後で実行時間に大きな変化は見られなかった．

4.4　考　　察

本節では，マイクロベンチマークプログラムでは効果が現れ，一般的なベンチマークプログラムでは効果が現れなかった原因についての考察を述べる．

3.2 節と 3.3 節で述べた影響がどの程度あるかを測定するために，単位時間当たりに関数オブジェクトを節約できた個数と，トップレベルへ移動した局所関数の単位時間当たりの呼び出し回数を数えた．関数オブジェクトを節約できた個数は，ある関数のトップレベルへ移動した局所関数の個数と，その関数の呼び出し回数から求めた．表 3 に結果を載せる．

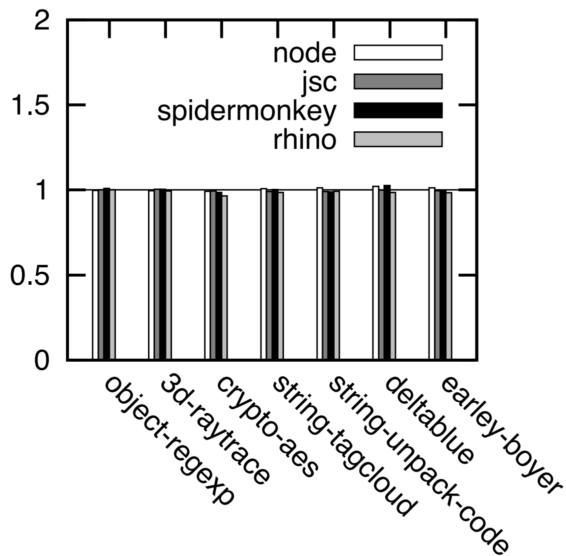


図 10　一般的なベンチマークにおける実験結果
Fig. 10　experimental results on standard benchmark programs

表 3　変換による影響の多寡
Table 3　some effect of program transformation

プログラム	関数オブジェクトを節約できた個数（毎秒）	移動した関数の呼び出し回数（毎秒）
object-regexp	0	0
3d-raytrace	0	0
crypto-aes	$3.2 * 10^3$	$1.5 * 10^3$
string-tagcloud	0	$1.3 * 10^3$
string-unpack-code	$1.4 * 10^2$	$8.0 * 10^{-1}$
deltablue	$4.7 * 10^{-1}$	$1.2 * 10^{-1}$
earley-boyer	$1.8 * 10^0$	$9.0 * 10^1$
マイクロベンチマーク	$3.0 * 10^6$	$2.6 * 10^6$

マイクロベンチマークプログラムと比べると，一般的なベンチマークプログラムでは，関数オブジェクトを節約できた個数・トップレベルへ移動した関数の呼び出し回数ともに少なく，多いものでもマイクロベンチマークプログラムの 1000 分の 1 程度だった．このことが，一般的なベンチマークプログラムで効果が現れなかった理由と考えられる．

5.　まとめと今後の課題

本研究では，Lambda Lifting の技法を JavaScript に応用し，その効果を測定した．マイクロベンチマークプログラムを用いた実験から，実行時情報を用いた

最適化を行う処理系と，静的なプログラム変換を組み合わせることで，実行効率の向上が期待できることを確認できた．

一般的なベンチマークを用いた実験では，どの処理系においても有意な効果が現れなかった．調査により，一般的なベンチマークでは，関数オブジェクトを節約できた個数および移動した関数の呼び出し回数の両方とも，マイクロベンチマークプログラムに比べて非常に少ないという結果が得られた．このことが一般的なベンチマークにおいて効果が現れなかった原因であると考えられる．

今後は，関数インライニングなど他のプログラム変換を試みる他，実験対象のプログラムを増やし，より広い範囲のプログラムに対して効果の有無を確認していく．また，実行時間だけでなく，メモリ消費量の変化など詳細なプロファイルについても調査する予定である．

質疑応答

Q:（寺田）表 3 で個数・回数ともに 0 はおかしい．局所関数をトップレベルへ持ち出すことによって，それに対応する関数オブジェクトが必ず作られてしまうので，節約できない場合があるのではないか．

A: 節約できない場合はありうるが，今回の実験では，オブジェクトの初期化などで一度だけ呼ばれる関数内の局所関数をトップレベルへ移動しているので，両方とも 0 で正しい．

Q:（荒川）クライアント側で動作する既存処理系をサーバ側で動作させた場合に起こりうる問題というのは考えられないか．

A: 考慮していなかった．今後調査する．

C:（荒川）処理系の性能を計るためのベンチマークプログラムだけでなく，node.js 上で動作するサーバアプリケーションなど，実アプリケーションにおいても実験を行ったほうが良いと思う．

Q:（滝本）Lambda-Lifting を選んだ理由は？

A: 関数型言語を対象としたプログラム変換を，JavaScript へ応用することを考え，まず古典的な手法である Lambda Lifting を選んだ．

Q:（滝本）インライン展開は考えていないのか？

A: 実装が今回は間に合わなかった．今後実験を行う．

Q:（滝本）静的な解析の結果を VM に渡すつもりか．

A: VM に渡し，実行時情報を用いた最適化に利用することを考えている．

Q:（鈴木）名前の衝突の可能性があるが，どう工夫しているか．

A: 特別な工夫はしていない．通し番号をふっている．

Q:（木山）サーバ側の処理系だから静的解析するということなのか？

A: サーバ側で事前にコンパイルを行う方式であれば，時間のかかる最適化を行っても良いだろうという動機でやっている．

Q:（倉光） V8 を実行速度で上回ることが最終的な目標なのか？

A: 実行速度の比較には GC などの実行時システムも関係するため，静的解析で違った価値を持たせたい．

Q:（倉光）Javascript は静的解析しにくいのでは．

A: ECMA Script 第 5 版で導入される strict モードのような，限定されたプログラムを対象にする．

Q:（川中）関数内に現れる局所関数のみを対象としているが，トップレベルにある無名関数も持ち出したほうが良いのではないか．

A: 現在は対象にしていないが，変換は可能である．

参 考 文 献

1) Andreas G. , Brendan E. , Mike S. , David A. , David M , Mohammad R. H. , Blake K. , Graydon H. , Boris Z. , Jason O. , Mozilla Corporation : "Trace-based Just-in-Time Type Specialization for Dynamic Languages", *Programming Language Design and Implementation* pp. 465–478, 2009
2) Craig C. , David U. , Elgin L. : "An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes", *Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications* pp. 49–70, 1989
3) Jong-Deok C. , Manish G. , Mauricio S. ,Vugranam C. S. ,Sam M. : "Escape Analysis for Java", *Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications* pp. 1–19, 1999
4) L. Peter D. , Allan M. S. : "Efficient implementation of the smalltalk-80 system", *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages* 1984
5) Thomas J. : "Lambda Lifting: Transforming Programs to Recursive Equations", *Conference on Functional Programming Languages and Computer Architecture*, pp. 190–203, 1985
6) ECMA Script Language Specification, `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`
7) Firefox, `http://mozilla.jp/firefox/`
8) Google Chrome, `http://www.google.co.jp/chrome/`
9) Node.js, `http://nodejs.org/`
10) Opera, `http://jp.opera.com/`
11) Rhino, `http://www.mozilla.org/rhino/`
12) Safari, `http://www.apple.com/jp/safari/`