Optimized Client-Server Distribution of Ajax Web Applications

Research Internship Report

Mehdi Bouaziz

École Normale Supérieure, Paris

mehdi.bouaziz@ens.fr

Abstract

There is a new wave in modern web 2.0 development based on unified frameworks of data manipulation and server-side programming as well as client-side programming with a single language, instead of JavaScript used with another server-specific language. MLstate's approach uses a single, functional language, named OPA, that is compiled to server code, database code and client code, and regroups all the layers of the application.

OPA compiler aims at doing all the logic of application functionality partitioning and security ensuring, allowing OPA developers not to focus and waste time on this complex and expensive work they were doing before. By automating these tasks, OPA get rid of sub-optimal partitioning and drastically reduce the cost of refactoring.

In this paper, we describe the automation of client-server code partitioning and propose algorithms to optimize it, which should result in faster, more responding web applications.

General Terms Languages, Performance, Security.

Keywords OPA, QML, MLstate, Web 2.0, Client-Server Partitioning, Slicing, Application Design.

1. Introduction

After years of research on virtual machine and application distribution, a fact revealed itself as a standard of web applications: (X)HTML, CSS, JavaScript, communicating with servers with protocol HTTP(S). Modern web applications relies on the fact that most browsers allow JavaScript to issue its own HTTP requests, with the use of the object XMLHttpRequest, a functionality used in the Ajax (Asynchronous JavaScript and XML) [Garrett, 2005] development approach. Ajax applications uses asynchronous communications between a server and a client, the user's browser. Code is run both on the server and on the client. In general, web application projects use different languages for client code (HTML, XML, JavaScript), server code (e.g. ASP.NET, C#, Java, PHP, Python, Ruby, ...) and data accesses (XQuery and many SQL languages). This diversity of languages creates a kind of "impedance mismatch" problem that needs to be solved in order to create efficient web applications.

MLstate eliminates this impedance mismatch by providing OPA[1], a unified language for all the three layers of web applications. Server code and data accesses are compiled into an executable server, whereas client code is translated into JavaScript. Unlike other unified languages, OPA developers do not have to elaborate manually the complex logic of client-server exchanges and the partitioning of its application into two separate sides. The partitioning is automatically done by the OPA compiler.

Safety and security are important additional concerns that need to be adressed. Indeed web applications are nowadays a critical part of our infrastructure, used for any service handling public as well as highly private information such as in banking, shopping, auctions, as well as emails, social networking, games, and certainly many other future online services. With OPA, safety and security are already well ensured by functional programming and typing. But a critical point which we have to focus on here is security in message passing on the network. One should be able to define easily its own security policy and expect the compiler to ensure it.

A developer rarely knows how to slice its code into client and server chunks, the problem may be particularly counter-intuitive and one would like developers to be able to focus on higher-level problems while automated methods are responsible for these kinds of optimizations.

In this paper we propose an automated answer to the slicing of mixed client-server OPA code. We start by describing a framework to statically reasoning on OPA programs. Then we introduce our main purpose, the slicer of OPA. According to discussed ways of optimization, we propose a model of client-server exchanges and finally optimizing algorithms.

MLstate'09 March 1-September 3, 2009, Edinburgh, Scotland, UK.

[1] OPA means *One Pot Applications*.

```
type person = { tel : string ; address : string }
val addressbook : person stringmap stringmap
val address_of_a_scott firstname = /addressbook/"Scott"/firstname/address

(* The path is first parsed as: *)
[ Id "addressbook" ; String "Scott" ; Id "firstname" ; Id "address" ]
(* and then transformed into: *)
[ Expr (Id "addressbook") ; Expr (String "Scott") ; Expr (Id "firstname") ; Field "address" ]
```

Figure 1: An example of the database preprocess on a QML code.

2. Static Analysis of QML Code

MLstate's web applications are written in OPA, which is then translated into QML. QML being more simple, we will work on QML abstract syntax tree (AST), but we may keep OPA syntax for code examples. Some restrictions will be needed to help static analysis.

2.1 The QML Language

QML is a functional language whose syntax is described in appendix of the joined report [Barbin and Bouaziz, 2009]. Its detailed semantics can be found in [Barbin, 2009]. Readers are strongly encouraged to get familiar with them in order to fully understand following sections.

Before implementing any static analyzer, a unified library for primitives [Barbin and Bouaziz, 2009] was needed to know some precise information on primitive functions used in a program in order to infer them on the whole program. These informations are:

- Implementation languages. For example database accesses or file I/O are strictly implemented in the server-side language whereas DOM accesses are only implemented in the client-side language (JavaScript). All non side-specific primitives should be implemented in both languages, but it is not compulsory ;
- Type and arity of functions. Where extern types[2] are used, serialization/unserialization joins must be added ;
- Side-effects and their *scope* of application. Even though QML is a functional language, some primitives do side-effects. Side-effects are mainly limited to database, file and DOM read/write. All three are independent scopes of side-effects and can be freely switched. Scopes of side-effects can be refined to more precise sub-scope (e.g. which file is being read or written) but it will usually require further static analysis like abstract interpretation (see [Bouaziz, 2008]). Another relevant information related to side-effects is their direction (read, write, or possibly both), which will be used in section 5.6.2.

[2] We denote here by an extern type a QML-abstract type which has no QML implementation but has an implementation in all back-end languages.

2.2 Restrictions

Static analysis of such a high-level functional language is quite difficult.That is why we need some restrictions on the input QML code to get a more simple AST. Actually all of these restrictions can be done, mainly by semantics-preserving rewritings.

2.2.1 Database Preprocess

Its purpose is to translate a database path of integers, strings and identifiers into a database path of expressions and field keys. This translation uses the database type schema. Figure 1 shows an example of this preprocess on a database path. We will need this preprocessing for typing and computing dependencies on identifiers.

2.2.2 Elimination of Overloads

All Overload nodes are eliminated from the AST and from the types. Reasoning with the semantics of overloads really brings difficulties. A pass of monomorphization on types and values transforms overloads into non-overloaded semantics-equivalent expressions.

2.2.3 Well Typed

As well as ensuring respect to the semantics and so ensuring compilation, typing (see [Benayoun et al., 2009]) will help us detecting functional values and inserting serialization-unserialization joins where a client-to-server or server-to-client exchange will be done.

2.2.4 Uniqueness of Names

If the same identifier is used in different scopes, then it is renamed by alpha-conversion. So we will not have to manage a local environment everytime the AST will be traversed. Better, we will be able to have a global environment with informations associated to each identifier, with no need to precise the scope which it refers to.

2.2.5 Elimination of `let type ... in`

This construction is only used by the typer to have abstract but locally non-abstract types. Once typing is done, we do not need this care of scope any longer and can freely get rid of these AST nodes.

2.2.6 Lambda-Lifting

According to [Johnsson, 1985], its aim is to eliminate free variables from local function definitions, including anonymous functions (whose a *fresh* name will be assigned to). It is done by adding an extra parameter for each free variable. After that, function definitions can be "lifted" to the top-level of the program. Thus each function has a unique top-level identifier. Moreover λ (or **fun**) can only be found as the first nodes of a top-level value, so that we can give an *arity* to each top-level function.

2.2.7 Elimination of let rec … in

Since **let rec** ... **in** must be followed by at least a lambda and lambdas have just been lifted, there is a free elimination of local recursive values. Recursive values are still present, but only as top-level functions (**val rec**). Static analyses often require fixpoint computation on recursive values. Then there is no more risk for it to be exponential in the level of imbrication of recursive declarations.

2.2.8 Elimination of Partial Applications

It can be useful to know when a function is "really" applied, that is to say, when its body will be executed. With functional languages and partial applications, it is not so easy. That is where *uncurrying* is needed. It is also helpful for compilation optimization [Dargaye and Leroy, 2009] and will be needed for compilation into JavaScript because some browser do not support partial applications [Resig, 2009]. Finally uncurrying functions simplifies the data-flow and control-flow graphs of the program.

2.2.9 Only Needed Recursion

Recursive values are often treated as a single group. If a group contains a function that is not really recursive with the other ones, this could lead the analysis not to be as good as expected. Not to lose any chance to have a better analysis it is worth eliminating non-needed **rec**. Thanks to uniqueness of names (see 2.2.4, above), this can be done easily by removing all **rec** and computing them from real cyclic dependencies.

3. The Slicer

After a brief description of *program slicing*, we show here the main subject of our paper, namely the *slicer* of OPA.

3.1 Program Slicing

According to [Harman and Hierons, 2001], originally introduced by [Weiser, 1984], program slicing is a technique for simplifying programs by focusing on selected aspects of semantics, usually one of the outputs of the program. Generally this is done by throwing away all statements having no effect on the computation of the selected value. Several techniques [Tip, 1995] exist and have different performance [Gold and Harman, 2007]. [Mastroeni, 2008] presents slicing as an abstract interpretation but unfortunately it cannot be applied on our slicing.

Most techniques are based on program dependency graphs [Reps, 1991] that permit to see the interesting statements as the predecessors of the selected values. Our approach will be based on such dependency graphs.

3.2 Our Slicer

Unlike normal slicing, our slicing does not aim at throwing away any statement. Actually its goal is, given a QML program, to split the program into two parts and choose which statements will be executed on client-side and which ones will be executed on server-side. Much more, as it can be done in automatic distribution of programs, statements can be duplicated to be executed on both sides, as long as side-effects are not done twice[3].

In the OPA compiler, the slicer is a rewriting pass preceeded by other passes among which those described in section 2.2. It gets as input the rewritten AST of the program and a list of server and client entry points. The server entry point is the entry point of the executable file resulting from the compilation. The client entry points are functions that will be executed on client-side under some client actions (like clicks). This is a list of top-level functions extracted from DOM events in the original OPA source code. Figure 2 shows an example of a part of an OPA program where the function get_latest_news is a client action. It shall be noticed that if div_news is called from the server, the function last_news will be called from server-side as well as from client-side.

```
last_news(subject) = /news/subject/last

update_news(subject) =
  [ #latest_news ←last_news(subject) ]

div_news(subject) =
  <a onclick={ update_news(subject) }>
    Refresh the latest news</a>
  <div id="latest_news">
    { last_news(suject) }</div>
```

Figure 2: A basic example of an OPA action.

The output of the slicer is the remaining QML server program which will be compiled by the QML-to-OCaml or the QML-to-LLVM compiler. To get that result, several intermediate steps are needed, whose outline is as follows:

1. Annotating and possibly rewriting the AST with side informations (client-only/server-only/both sides).

[3] Actually, it will be pointless to check this assertion as long as we have only side-specific side-effects.

2. Rewriting the AST so that every function will be executed on one single side and so that calls will always be directed from the client to the server.
3. Generating URLs for server-side functions called by the client; these functions are new server *web*-entry-points.
4. In client-side functions, replacing calls to server-side functions by AJAX requests to the right URL and inserting serialization/unserialization joins.
5. Compiling client-side code to JavaScript and placing it in literal string values on the server.

Step 1 is the core part of the slicer and will be treated in parts 4 and 6. Steps 2 to 5 will not be detailed here.

3.3 Discussions

We propound here some important assumptions for the rest of the paper in order to simplify some details to the reader.

3.3.1 Granularity

An important parameter in slicing is its granularity, that is to say the size of chunks of code that will be considered atomic. As any statements of QML may have to be placed on a certain side, our granularity cannot be less precise than the statements.

However we want to keep the structure of the user program, as we think that a function is an essential cutting already done by the user. So our slicing will not enter calls. Function inlining would result in a more precise slicing but inlining can be exponential in the size of the initial program. Then, in the rest of the paper we will equally refer to programs as to functions.

Thus we will consider top-level functions as primitives available on client-side as well as on server-side. Actually each top-level function will be replicated on both sides and optimized according to these input/output side-constraints, resulting in two versions of the function with different cost[4]. A call to a given function will have different cost depending on its side. Similarly side-specific primitive functions can be seen as two version of the same function, one having a null cost and the other having an infinite cost.

In case of higher-order functions, the cost will be parametric in the cost of its functional arguments. In a call, costs of its arguments can be approximated by a worst-case or an average cost. As a simplification for understanding, we will assume in the rest of the paper that function calls are like primitive function calls of a null cost. But let us keep in mind that taking this into account will not significantly change any result or algorithm except that the resulting slicing may be cheaper in terms of exchanges.

3.3.2 User Annotations

OPA developers are free to annotate statements of their programs with *user side-annotations* to enforce them to be executed on a specific side, or enforce them to be replicated on both sides.

3.3.3 Security Annotations

Statements can also be annotated with tainting informations. Values can be marked to be ς-*black*, ς being either Client or Server. ς-black value should never be seen on the ς-side. Neither its dependencies should be seen on the ς-side.

Functions can be marked to be *whitening*. Dependencies of a call to a whitening function will not be marked black. For instance, some password can be marked client-black and a crypt function can be marked whitening to enable crypted messages to be exchanged between server and client.

These tainting informations are resolved to pre-computed sides as well. This resolution is a kind of typing or an abstract interpretation ([Cousot, 1997]).

3.3.4 Partitioning is Not Distributing

Our slicer will not parallelize executions of programs. Server calls will be synchronous and the execution will be on a single side at any time. Further optimizations could be done by asynchronous calls and parallelization [Hunt and Scott, 1999]. But such optimizations are much more complex and make much harder the management of the control flow, and are left as a future work.

3.3.5 Model of the Server

To keep the functional structure of the language, the server is a state-free server in the sense that, except possibly in the database, the server is not keeping a trace of the client. So environments must be passed when needed.

We will not discuss here performance issues as it can be solved with cache techniques.

Another important point is that clients do not interfere with each other's performance. The server is considered to be ideal with infinite memory and as much as CPUs as needed. In case of heavy load, the server can be distributed [Cardellini et al., 1999].

4. Directions of Optimizations

What does "Optimized" means in the title of this paper ?
What is the value to be optimized ?
How can it be computed ?

On the other hand, we will see in section 6 how the value which we will call the *cost* can be optimized.

4.1 The Ideal Optimization

Actually, the best result expected is the fastest reactivity of the application from the viewpoint of the user. More and more, the user is eager to get the response as quick as in non-web applications [Nielsen, 1994]. That is to say, we want him to have the most immediate result to each of its action (like clicks) everytime a reaction is expected.

One may also want to reduce the load of the server in order to support more clients. It may be a researched goal

[4] The cost will be defined in sections 4.3 and 5.4.1.

for low-cost websites. According to section 3.3.5, we will not. But we think one's optimization can be based on our approach.

4.2 Difficulties

There are a lot of barriers to take this time of reaction as the value to optimize, starting with its measuring, mainly due to the randomness of the parameters:

- the diversity of *clients*, hardware and web browser, including the rendering time of the web browser and its JavaScript processing time ([RockStarApps, 2007], [Kıcıman and Livshits, 2007]): the CPU(s) and its load, the web browser and its rendering engine, ...; the client network capacity and its load
- the *server* CPU(s) and network loads
- the vagaries of the *network* and the Internet

Regardless of the establishement of a model of this time, not very relevant because of its high variability, we are faced to the variety of size of exchanges, going from a single boolean to a big file. Whereas time analysis is mostly based on a worst-case approach, we would rather need a static analysis of a time depending on dynamic data.

4.3 Chosen Optimization

In view of these difficulties, we chose to start with the most significant sources of time consumption. Without considering items that cannot be part of the slicer, according to [Theurer, 2006], [Yahoo!, 2008] and [Zyp, 2008], we should focus on:

1. first, reducing the number of HTTP requests (statically known)
2. and then, reducing the size of these requests (either statically or dynamically known)

A request involves a network communication which is higly expensive. For typical requests (less than 1 KB), doubling the number of requests doubles the waiting time of the client and the server load whereas doubling the size of a request has no significant impact.

Of course, most of the time, it is preferable to have two requests of 10 KB each rather than a single one of 100 MB. We thought yet that this case would not often appear and that reducing requests would not result in such an excessive increase of their size.

So choosing the number of client-server exchanges as our optimization criterion seems good enough. More precise measures can be done afterwards but will generally require dynamic choices.

5. A Model of Client-Server Exchanges

In this section we will try to find a model which we will be able to reason on in order to achieve the task of the slicer. Our approach is to go from the expected result to our final model.

5.1 The Expected Result

We recall that our slicer will have to chose for each statement of the program which side(s) it will be executed on. Each statement will be annotated with a value within the set $\{C, S, CS\}$ [5]. Such a choice on the program can be observed by colouring the source code with three different colours. Figure 3 shows an example of such a code with a valid slicing. Reading the number of client-server exchanges on this graphical representation is not very comfortable.

```
whole : string // a persistent string

append_msg() =
  new_whole = /whole ^ #msg /* compute the
      new whole text by concatenating the old
      whole text and the new message */
  /whole = new_whole // update the database
  #whole = new_whole // update the client display

some_html = <input type="text" id="msg" />
  <a onclick={ append_msg() }>
    Append my message</a>
  <div id="whole">{ /whole }</div>
```

Figure 3: Example of a program coloured with respect to side annotations. C is green, S is red and CS is blue.

Let us physically separate the code into two sides but with keeping the control flow unchanged so that side-changing flow naturally appears (Figure 4). Now the parameter of our optimization is readable as the number of arrows.

```
c_msg = #msg
                    ⟶
                      new_whole = /whole ^ c_msg
                      /whole = new_whole
                    ⟵
#whole = new_whole
```

Figure 4: Sides-separation of the function append_msg with visible exchanges.

5.2 Control and Data Flow Graphs

Actually this representation can be seen as an abstraction of the dependency graph of the program, where some edges are relevant to our optimization. More formally, we define the *control and data flow graph* (CDFG) of a QML program P as:

Definition 1. $cdfg(P) = (V, E)$ where $V = \{v | v$ is a node of the AST of $P\}$, and $E = \{(v, w, t) | v, w \in V$ and either

[5] C stands for Client, S for Server and CS for both.

$t = Data$ and computation of w directly depends on data v, or $t = Control$ and w must be executed after v}.

An equivalent definition of $cdfg(P)$ can be given with the 3-tuple $(V, E_{Data}, E_{Control})$ where $E_t = \{(v, w)|(v, w, t) \in E\}$ for t in $\{Data, Control\}$.

A node w *directly depends* on v if v is to be used in the computation of w, i.e. a subnode or an identifier. A statement w *must be executed after* v if w does side-effect on a scope $s_w \subseteq \mathcal{S}$, v does a Write side-effect on a scope $s_v \subseteq \mathcal{S}$ and $s_w \cap s_v \neq \varnothing$, where $\mathcal{S}$ is the set of side-effects scopes. Notice that Read side-effects does not depend on each others.

5.2.1 Scopes of Side-Effects

As discussed in section 2.1, scopes of side-effects are an abstraction of the semantics of primitives. The roughest approximation is to consider only one possible scope by taking $\mathcal{S}_\top = \{\top\}$. Defining all statements as doing a Read-Write side-effect on $\{\top\}$ will enforce the order of statements.

Because different functions involve different scopes, it is very easy to have a little more precise set:

$$\mathcal{S}_0 = \{Database, FileSystem, DOM\}$$

This set has the following interesting property that each scope is dedicated to one side, which guarantees that they will not be replicated on both sides and applied twice.

With further refinement, e.g. by abstract interpretation, we could have a more precise set:

$$\mathcal{S}_\sharp = \bigcup_{p \in DB^\sharp} DB_p \cup \bigcup_{p \in FS^\sharp} FS_p \cup \bigcup_{p \in DOM^\sharp} DOM_p$$

where $DB^\sharp$, $FS^\sharp$ and $DOM^\sharp$ are (abstract) sets of database paths, file system paths and DOM paths.

5.2.2 Side-Choice

To build the side-separated code, we need an extra information which is the side annotation on each node:

Definition 2. A *side-choice* $\sigma : V \to \Sigma$ for the program P is a function that maps each node v of its CDFG onto a local side-choice. With $\Sigma = \{\{C\}, \{S\}, \{C, S\}\}$.

Given a program P, its CDFG $cdfg(P) = (V_0, E_0)$ and a side-choice σ for P, we can define a *control and data flow and side graph* (CDFSG) of P, which is just a side-annotated CDFG where both-side-annotated nodes are duplicated:

Definition 3. $cdfsg(P, \sigma) = (V, E)$ where $V \subseteq V_0 \times \{C, S\}$ is defined as $V = \{(v_0, \varsigma)|v_0 \in V_0 \text{ and } \varsigma \in \sigma(v_0)\}$ and $E = \{(v, w, t)|v = (v_0, \varsigma_v) \in V, w = (w_0, \varsigma_w) \in V, (v_0, w_0, t) \in E_0, \text{ and } \sigma(v_0) = \{C, S\} \Rightarrow \varsigma_v = \varsigma_w\}$. That is to say we do not add useless exchanges in case of duplicated statements, as shown in Figure 5.

Figure 6 shows the annotated CDFSG where Figure 4 can be derived from.

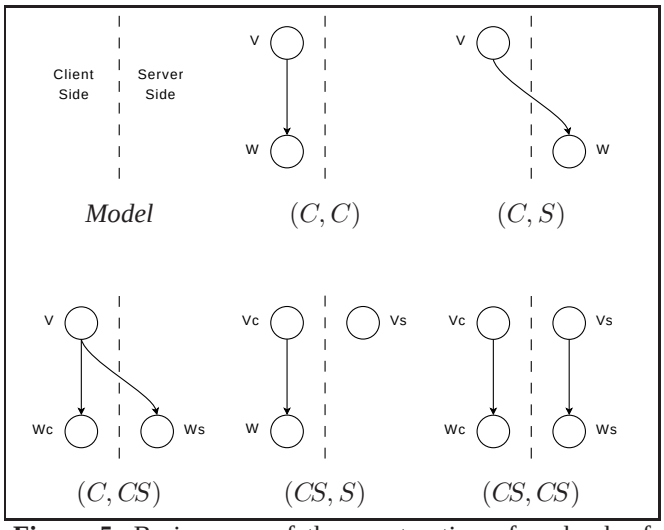

Figure 5: Basic cases of the construction of a chunk of CDFSG from the CDFG and two local side-choices $(\varsigma_v, \varsigma_w)$. Case (S, S) is similar to (C, C), (S, C) to (C, S), (S, CS) to (C, CS), and (CS, S) to (CS, C).

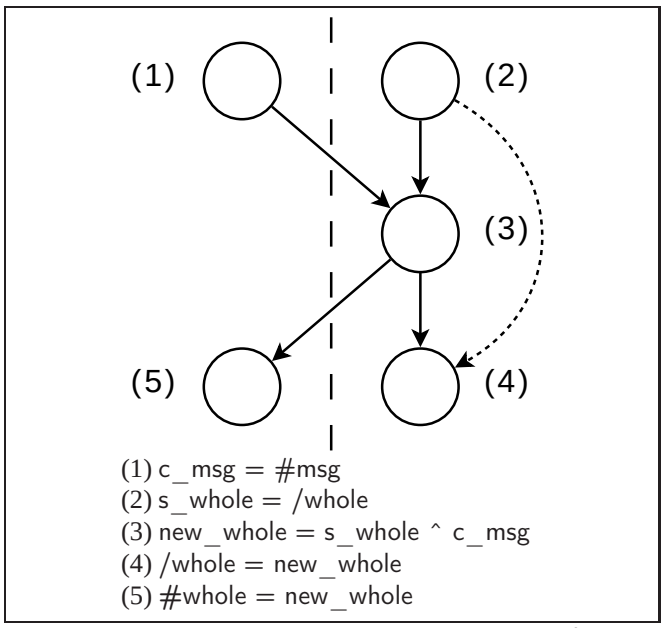

Figure 6: Dependency Graph of the function append_msg. The dotted arrow represents a Control edge whereas normal arrows represent Data edges.

5.3 Pre-Side-Choice

A side-choice is the result of optimization, described in section 6. So, before having a side-choice, we only have partial informations on sides, like sides of some primitives or any other expression explicitly annotated by the user. Let us define these partial informations as:

Definition 4. A *pre-side-choice* $\sigma : V \to \Sigma_0$ for the program P is a function that maps each node v of its CDFG

onto a local pre-side-choice. Σ_0 is the set of local pre-side-choice: $\Sigma_0 = \{C, S, CS, A, C^*, S^*, A^*\}$.

C means that the statement will be placed on client-side and possibly on server-side too. So S means for server-side. CS means it will be placed on both sides whereas A means it can be placed on any side. A star means that the statement cannot be duplicated. Notice that CS^* would be equivalent to CS.

We usually get pre-side-choice values in the set $\{A, C^*, S^*\}$ but for the sake of genericity we extended this set to Σ_0 defined above.

Σ_0 is a partially ordered set, more precisely a meet-semilattice, where A is the infimum and $\Sigma_0^m = \{C^*, S^*, CS\}$ the maximal elements (Figure 7). For $s_0, s_1 \in \Sigma_0$, $s_0 < s_1$ if s_1 respects the definition of s_0 and is more precise than s_0.

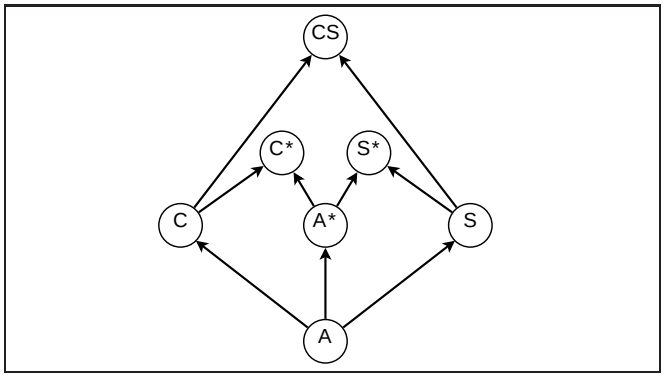

Figure 7: Hasse diagram of the semilattice Σ_0.

Fact 5. There is a canonical bijection between Σ and the maximal elements of Σ_0:

$$\begin{array}{rccc} \rho: & \Sigma_0^m & \rightarrow & \Sigma \\ & C^* & \mapsto & \{C\} \\ & S^* & \mapsto & \{S\} \\ & CS & \mapsto & \{C, S\} \end{array}$$

So a pre-side-choice σ so that $\forall v \in V$, $\sigma(v) \in \Sigma_0^m$ can be *promoted* to a side-choice $\sigma' = \rho \circ \sigma$.

5.4 Control and Data Flow and Pre-Side Graphs

Given a program P, its CDFG $cdfg(P) = (V_0, E_0)$ and a pre-side-choice σ for P, we can define a *control and data flow and pre-side graph* (CDFPSG) of P, which is a pre-side-annotated CDFG where both-side-annotated nodes ($\sigma(v) = CS$) are duplicated:

Definition 6. $cdfpsg(P, \sigma) = (V, E)$ where $V \subseteq V_0 \times \Sigma_0^\vee$ is defined as $V = \{(v_0, \varsigma) | v_0 \in V_0$ and $\varsigma = \sigma(v_0)$ if $\sigma(v_0) \neq CS, \varsigma \in \{C^*, S^*\}$ if $\sigma(v_0) = CS\}$; $\Sigma_0^\vee = \Sigma_0 \backslash CS$, and $E = \{(v, w, t) | v = (v_0, \varsigma_v) \in V$, $w = (w_0, \varsigma_w) \in V$, $(v_0, w_0, t) \in E_0$, and $\sigma(v_0) = CS \wedge \sigma(w_0) \in \Sigma_0^m \Rightarrow \varsigma_v = \varsigma_w\}$. We keep side-duplication of dependencies as long as sides are not fully decided (i.e. $\sigma(v_0) \notin \Sigma_0^m$).

The goal of the slicer will be: given a CDFPSG of a program P, get its pre-side-choice more precise to promote it up to a side-choice (which will give a CDFSG of P). However we need a restriction on the form of P:

Proposition 7. Given a program P, the following propositions are equivalent:

1. The CDFG of P is a *directed acyclic graph* (DAG).
2. Any CDFPSG of P is a DAG.
3. Any CDFSG of P is a DAG.
4. P has no recursive function.

Proof.
- (1) $\Rightarrow$ (2). Use as the pre-side-choice the constant function equal to A. Any other CDFPSG of P can be obtained by side-refinement (see Definition 9 below).
- (2) $\Rightarrow$ (3). A CDFSG of P can be obtained by promotion of a CDFPSG.
- (3) $\Rightarrow$ (1). A CDFSG of P is obtained from its CDFG by at most duplicating some nodes and edges, so it does not change its acyclic property.
- (1) $\Leftrightarrow$ (4). It is ensured by the restriction of section 2.2.9. □

In the rest of this section, we will assume that we are treating only non-recursive functions, so any CDFSG will be a DAG. We will tackle recursion in section 5.6.

5.4.1 The Cost Function

From a CDFSG of a program, computing the cost of a side-choice is immediate in terms of number of client-server exchanges:

Definition 8. The cost of a side-choice is given by:

$$\kappa(P, \sigma) = \text{card}\{(v, w, Data) \in E | \varsigma_v \neq \varsigma_w\}$$

5.5 Atomic Operations

Given a CDFSG, there are a lot of possible ways to form valid semantically-equivalent programs, as shown in Figure 8. Indeed the CDFSG is a DAG, that is to say a partial order. We add to the graph a *total order* $\theta(P, \sigma)$ on statements, which must include the partial order of the CDFSG. This order can be described as an ordered list of vertices of the graph $(v_0, v_1, v_2, \ldots)$ or as a set of edges of a new type $Order$. We name a pair $(cdfsg(P, \sigma), \theta(P, \sigma))$ an *ordered CDFSG*. To an ordered CDFSG corresponds a unique program.

Identically, a total order can be added to a CDFPSG to form an *ordered CDFPSG*.

Definition 9. We then introduce some *atomic operations* that can be performed on an ordered CDFPSG $cdfpsg(P, \sigma) = (V, E)$ and θ, without changing the semantics of the program:

- *Side-refinement*. Given a vertex $(v_0, \varsigma) = v \in V$. If $\varsigma \notin \Sigma_0^m$ then we can chose $\varsigma' \in \Sigma$ so that $\varsigma < \varsigma'$ and update σ and the graph as a consequence.

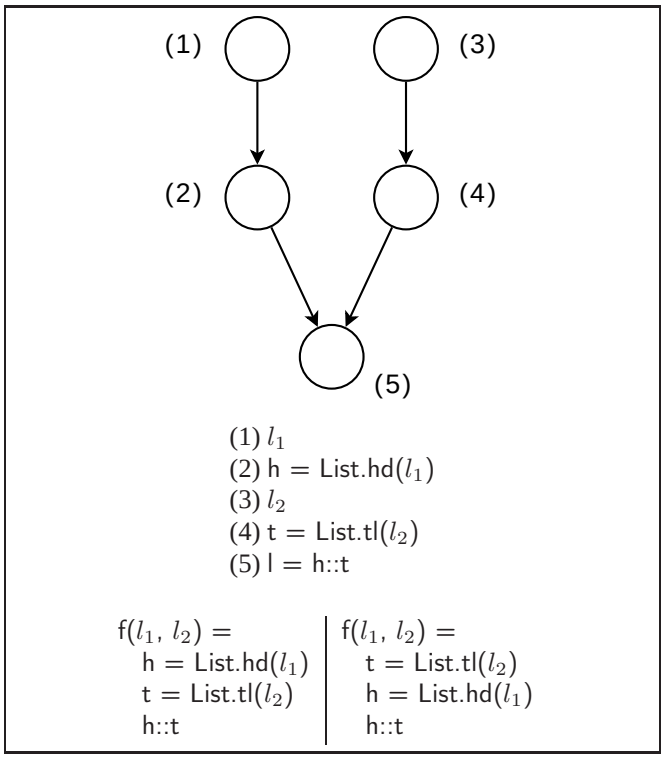

Figure 8: Different semantically-equivalent reconstitutions of a unique CDFSG.

- *Reordering*. The ordering θ of vertices can be changed as long as it remains compatible with the partial order given by the graph.
- *Grouping*. Given two edges $(v, w, t), (v', w', t') \in E$ so that $t = t'$, $\{\varsigma_v, \varsigma_w, \varsigma'_v, \varsigma'_w\} \subseteq \{C^*, S^*\}$, $\varsigma_v = \varsigma'_v$, $\varsigma_w = \varsigma'_w$, and $max(v, v') < min(w, w')$ according to the ordering θ[6], then we can group the edges by boxing v and v' into a pair, the new vertice u with $\sigma(u) = \varsigma_v$, u is replicated as u' with $\sigma(u') = \varsigma_w$ and will be unboxing into w and w'. A placement in θ must be chosen for u and u' so that $max(v, v') < u < u' < min(w, w')$. Grouping will be called *useful* if $t = Data$ and $\varsigma_v \neq \varsigma_w$. Figure 9 shows an example of grouping.

Excepting side-refinement, these atomic operations can also be defined on CDFSG.

After a certain number of atomic steps, we can arrive to a state where our pre-side-choice can be promoted to a side-choice, our CDFPSG to a CDFSG from which the cost of our side-choice can be computed. Whereas reordering and grouping are optional, side-refinement is necessary unless the original pre-side-choice was promotable to a side-choice.

Proposition 10. Optimizing the cost κ of a program using atomic operations is decidable.

Proof. Let us show that the optimal solution is in a finite set of states. Atomic operations can be applied in the order Ordering, Side-refinement and finally Grouping, without losing any generality.

1. Fix the initial ordering. Since there is a finite number of such initial ordering, all remaining steps will have to be done for each ordering.
2. Fix the side-choice by side-refinements. Number of ways to do it is bound by $6^{|V|}$ where $|V|$ is the number of vertices.
3. We now have a CDFSG and can compute the cost that we call κ_0 which is a finite non-negative integer. By induction on κ, and using lemmas 11 and 12, we prove that by useful groupings only with get to a no-more-optimizable state. Lemma 12 tells us that non-*useful* groupings are useless, while they can lead to non-ending (useless) groupings.

□

[6] Thus there is no path from w to v' or from w' to v.

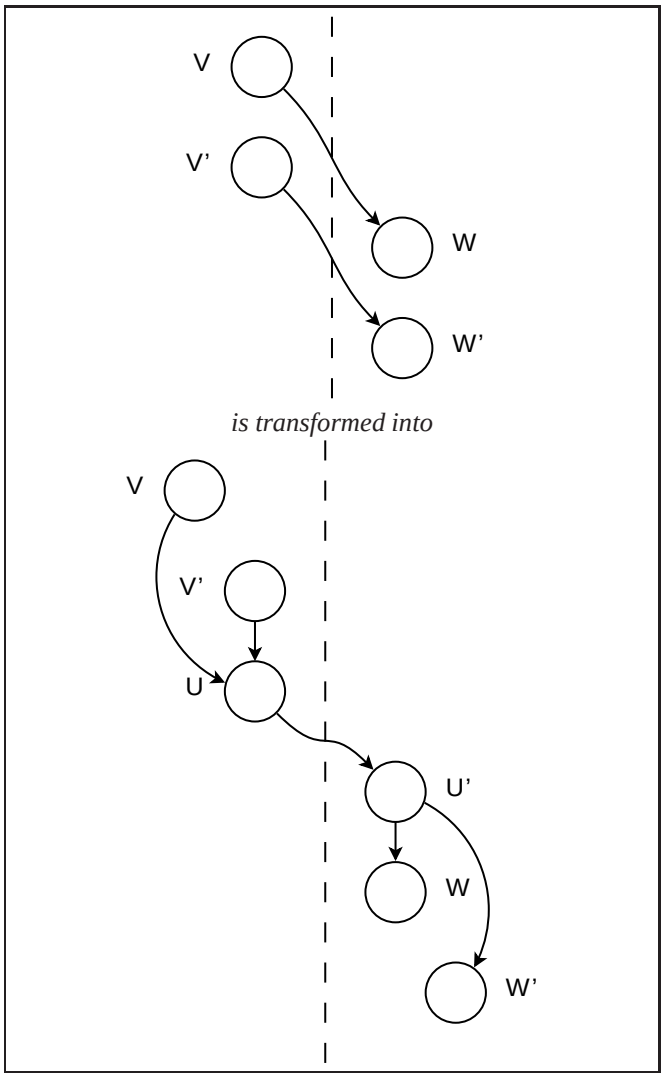

Figure 9: An example of edge grouping. The first exchange is delayed / the second exchange is moved forward.

Lemma 11. In a CDFSG (V, E), the number of useful groupings is bound by $\binom{\kappa}{2}$.

Proof. The number of useful groupings is bound by the manner of chosing two distinct edges in E, each edge (v, w, t) being so that $t = Data$, $\varsigma_v \neq \varsigma_w$, that is we just ignore the ordering constraint. □

Lemma 12. In a CDFSG, a useful grouping makes κ decrease by 1 and a non-useful grouping let κ unchanged.

Proof. Consider a grouping of $(v, w, t), (v', w', t') \in E$. If this grouping is useful then $t = t' = Data$, $\varsigma_v \neq \varsigma_w$ and $\varsigma'_v \neq \varsigma'_w$. Both edges count as 1 in κ. They are replaced by five edges $(v, u, Data)$, $(v', u, Data)$, $(u, u', Data)$, $(u', w, Data)$, $(u', w', Data)$ with $\varsigma_u = \varsigma_v$ and $\varsigma'_u = \varsigma_w$. So only the third one will be counted in the new κ.

If this grouping is not useful then either $t = t' = Control$ or $\varsigma_v = \varsigma_w = \varsigma'_v = \varsigma'_w$. Both edges were not couting in κ. So, either newly added edges are of type $Control$ or on the same side. So κ remain unchanged. □

Finding a better algorithm is discussed in section 6.

5.6 Handling Recursion

Since there is no loop construction in QML, recursion is the only way to do loops. [Chong et al., 2007] approximates loops by considering them doing exactly 10 iterations. However recursion is much more general than loops and so is harder to get along with.

We classify recursive functions in groups with different level of difficulty:

5.6.1 Simple Recursive Functions

The simple case is when we can put the function on one side only.

Given a CDFPSG of the function, if the pre-side-choice takes its values in the sublattice $\Sigma_0^C = \{A, A^*, C, C^*\}$ of Σ_0 or in the sublattice $\Sigma_0^S = \{A, A^*, S, S^*\}$ then the whole function can be placed on the client-side in the first case, on the server-side in the other one. The pre-side-choice of the function can be easily promoted a side-choice to $\{C\}$ or to $\{S\}$.

The cost of the function will be 0, as it does not have any client-server exchange.

5.6.2 More Complex Recursive Functions

The function is in that case if it has mixed sides (ς_0 and ς_1 so that $\{\varsigma_0, \varsigma_1\} = \{C, S\}$) and vertices on ς_0 are only statements doing Write side-effects (remember that there is no Read side-effect on the same scopes). Then we can place the whole function on side ς_1 and delay writes to the end. Figure 10 shows an example of such a transformation. This transformation can be seen as reordering and grouping the overall "loop".

The cost of the function will be 1, as it has a single exchange, from ς_1 to ς_0.

```
values : int list

// The function:

rec display_and_sum(i) =
  x = /values/i
  #values[i] = x
  if i > 0 then
    x + display_and_sum(i − 1)
  else
    x

// Is transformed into:

rec aux(actions, i) =
  x = /values/i
  actions = (#values[i] ←x)::actions
  if i > 0 then
    (actions, sum) = aux(actions, i − 1)
    (actions, x + sum)
  else
    (actions, x)

display_and_sum(i) =
  (actions, sum) = aux([], i)
  exec_actions(actions)
  sum
```

Figure 10: Delaying writes at the end of a function.

5.6.3 The Other Recursive Functions

Other cases may not be tractable as shown in Figure 11.

```
rec f(i) =
  if /db/i <= 0 then
    0
  else
    1 + f(#u[/db/i])
```

Figure 11: A too complex resursive function.

For these cases, we can only propose a fallback solution which is to consider the function as non-recursive and optimize it so. That is optimizing each iteration of a loop without considering it as a whole. The user will be warned and invited by the OPA compiler to rewrite his function into a better way, if possible.

6. Main Algorithm

This part is dedicated to the first step of Section 3.2, that is the annotation of the program AST with side choices, according to the optimization described in section 4.3.

6.1 Without Optimization

Before trying to optimize the slicing, let us see what is going on when we do not try to optimize. The easiest way to do it is to place everything on client-side and build a server atomic request everytime a server call is needed (database accesses or other server-specific primitive).

In terms of atomic operations, it means only side-refining in a client-oriented way ($A, C, A^*, C^* \mapsto C^*$; $S, S^* \mapsto S^*$; $CS \mapsto CS$).

Of course it works, but we can see that, even with simple applications, some actions generate several server requests where any human being could have put only one.

6.2 Side-Refinement

If we did not know what statements produce side-effects, we could not be able to reorder them and should keep the order decided by the user. This is the least optimization we can expect from the slicer. Let us first have a look at how optimization can be performed in this case.

To a very similar problem, [Chong et al., 2007] propose to solve it by integer programming:

Given a program P, its CDFG $cdfg(P) = (V_0, E_0)$, a pre-side-choice σ and the corresponding CDFPSG (V, E), the problem of optimizing side-refinements is expressed as an instance of an integer programming problem.

A solution to the problem assigns all variables in the problem a value in $\{0, 1\}$. We associate to each vertex $v = (v_0, \varsigma_v) \in V$ two variables s_v and c_v. $s_v = 1$ if the statement v is replicated on the server, 0 if not. So is c_v with client-side. Table 1 shows constraints used to ensure consistency wrt. the pre-side-choice.

ς_v	constraints	comments / degree of freedom	
A	$c_v + s_v \geq 1$	ensures at least one side	2
C	$c_v = 1$	we can only play on s_v	1
S	$s_v = 1$	we can only play on c_v	1
A^*	$c_v + s_v = 1$	c_v and s_v are directly linked	1
C^*	$c_v = 1, s_v = 0$	fully constrained	0
S^*	$c_v = 0, s_v = 1$	fully constrained	0
CS	$c_v = 1, s_v = 1$	fully constrained	0

Table 1. Constraints on statements according to the pre-side-choice.

We associate to each edge $e = (v, w, t) \in E$ two variables κ_e^{cs} and κ_e^{sc}. $\kappa_e^{cs} = 1$ if $t = Data$, $s_w = 1$ and $s_v \neq 1$, that is to say w will be executed on server-side but v will not. Hence a constraint $\kappa_e^{cs} \geq s_w - s_v$. Symmetrically we have $\kappa_e^{sc} \geq c_w - c_v$.

The purpose is to find an assignment to all variables that satisfies all constraints and minimizes the cost

$$\kappa = \sum_{e \in E} (\kappa_e^{cs} + \kappa_e^{sc})$$

It shall be noticed that this definition is equivalent to our first definition of the cost (Definition 8 in section 5.4.1).

[Chong et al., 2007] claim this integer programming problem is polynomial-time solvable because this particular problem has the property that its linear relaxation[7] always has an integral optimal solution which is the optimal solution to the integer programming problem, and linear programming problems are polynomial-time solvable.

They also give a lead to a more efficient algorithm by reducing the integer programming problem to an instance of the maximum flow problem on a graph derived from the CDFPSG of the program, resulting in an algorithm running in $O(V^3)$ where V is the number of statements, but in pratice highly improved by heuristics.

6.3 Reordering and Grouping

Adding reordering blows up the combinatorial complexity of the problem. However we can apply reordering and grouping on the result of the previous algorithm.

The key idea is to form blocks of several vertices instead of atomic groupings of edges. A set $B \subseteq V$ of vertices can be grouped to form a *block* if all vertices are on the same side ς_B, and for all $v_0, v_1 \in B$, there is no path $v_0 \to w_0 \to \ldots \to w_n \to v_1$ so that $\exists\, 0 \leq i \leq n, \varsigma_{w_i} \neq \varsigma_B$, i.e. no statement of the block depends on a statement on the other side which depends on another statement of the block.

Let us build a covering set $\mathcal{B} = \{B_0, B_1, \ldots, B_n\}$ of blocks, i.e. $\bigcup_{0 \leq i \leq n} B_i = V$ and $\forall\, 0 \leq i < j \leq n$, $B_i \cap B_j = \varnothing$. A covering set of blocks of maximal size can be obtained by exploring the graph in a topological order remaining on the same side and aggregating vertices as long as they do not break the building rules of a block. Once no vertices can be added to a block, a new one is started, and so on. Then we have the following property:

Proposition 13. $\forall 0 \leq i < j \leq n, \varsigma_{B_i} \neq \varsigma_{B_j}$ implies that all Data edges between B_i and B_j forms useful groupings.

Proof. Indeed, by construction of blocks, there exists a valid total order so that either $B_i < B_j$ (in the sense that $\forall v, w \in B_i \times B_j, v < w$, i.e. $sup\, B_i < inf\, B_j$), or $B_i > B_j$. □

Given two distinct blocks, we can group edges between them into a single edge.

6.4 Sum Up

The process of the slicer in the OPA compiler can be summed up in the following steps:

1. Applying rewritings on the AST (Section 2.2).
2. Gathering user side-annotations (Section 3.3.2).
3. Gathering user security-annotations and resolving them as user side-annotations (Section 3.3.3).
4. Processing higher-order functions (Section 3.3.1) and recursive functions (Section 5.6).

[7] The linear relaxation of the problem is obtained by replacing the constraint $s_v, c_v, \kappa_e^{cs}, \kappa_e^{sc} \in \{0, 1\}$ with $s_v, c_v, \kappa_e^{cs}, \kappa_e^{sc} \geq 0$.

5. Building a pre-side-choice (Section 5.3) from user side-annotations.
6. For each function, Building its CDFPSG (Section 5.4) from the AST and the pre-side-choice.
7. And Refining sides (Section 6.2).
8. And Reordering and Grouping (Section 6.3).
9. On the reassembled program, applying steps 2 to 5 from Section 3.2.

6.5 Results

Although we do not have any measurement on our main goal, that is the time of reaction of our applications, tests from our battery were all well sliced and optimized (or we really *thought* it was optimized, we are just human beings) as expected. Our tests are all either simple stupid tests or real pratical examples (which were unfortunately written by developers conscious of well written code and performance).

The algorithm is efficient enough to compile real applications thanks to the separate optimization of each function.

7. Related Work

Several languages have been recently proposed for programming web applications in a unified way.

Ocsigen ([Balat, 2006]), HOP ([Serrano et al., 2006]) and Links ([Cooper et al., 2006]) are all functional languages but do not have any automated slicing mechanism. Sides must be written by the developer.

Hilda ([Yang et al., 2007]) is based on UML and the relational data model. Programs have a hierarchical structure and so are easier to slice that more expressive languages. They also formulate the optimization of their partitioning as an integer programming problem. Their cost function is more precise but the solving of partitioning is NP-hard. They use an approximation to solve it efficiently.

Swift ([Chong et al., 2007]) is a Java-like language which aims at building web applications that are secure by construction. Side computation is based on security annotations of each statement. We based our algorithm on theirs.

Doloto ([Livshits and Kıcıman, 2008]) is not a language but a tool analyzing workload of web applications and rewriting to introduce dynamic code loading. Its goal is the same as ours but Doloto can only optimize the loading of the application, not its cruising reactiveness.

8. Conclusion and Future Work

We developped a framework for static analysis of OPA programs. We proposed a model of client-server exchanges of an OPA program, based on dependency graphs. We formalized the partitioning problem and suggested algorithms to solve it efficiently.

Our slicer is a part of the OPA compiler, written in OCaml. It is able to give a valid slicing to any OPA program according to user side and security annotations, and has some heuristics for optimizing recursive programs.

Client-server slicing of web applications still can be improved by refining cost, granulary and better taking variable parameters into account. For example we can differentiate clients' performance: help PDAs and mobile phones by putting heavy computations on server-side. Refinement can be inspired from refinement techniques of program slicing, such as splitting records or constructing dynamic slicings, like [Amiri et al., 2000].

Independently from our slicer, page and client code dynamic loading sur as Doloto can be added to our system.

An effort can be done on time analysis and introduction of results from abstract interpretation.

Speed of applications may also be improved by parallelizing execution on client and server instead of keeping the synchronous flow of the program.

In front of all these ways of further research, we are optimistic for future web applications development.

Acknowledgments

I really would like to thank Henri Binsztok who proposed me this internship with a subject full of stimulating challenges in a wonderful place of the world[8].

Thanks to Louis Gesbert for his plaisant company and having made me making the most of the Edinburgh festival.

Thanks to Rudy Sicard who supported me from the beginning to the end of this internship and rewieved this report.

Of course, I cannot forget Mathieu Barbin who had to put up with me...

Finally I am so grateful to Alice who more than helped me when I really needed.

References

[Amiri et al., 2000] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 307–322.

[Balat, 2006] Vincent Balat. Ocsigen: typing web interaction with objective Caml. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 84–94, New York, NY, USA. ACM.

[Barbin, 2009] Mathieu Barbin. Semantics & Reference Implementations for a Functional Language with Overloads, Extensible Records and Parallel Evaluation. Technical report, MLstate.

[8] Some quotations from a wall in the Edinburgh airport: "It possesses a boldness and grandeur beyond that I have ever seen", Thomas Pennant ; "Edinburgh is a city which encourages you to think about what a city is", Murdo Macdonald ; "Edinburgh is a dream of great genius", Benjamin Haydon ; "This dream in masonry and living rock is not a drop scene in a theatre, but a city in the world of reality", Robert Louis Stevenson ; "There is no habitation of human being in this world so fine in its way as this, the Capital of Scotland", Andrew Carnegie ; "This is one of the loveliest cities in the world", Yehudi Menuhin.

[Barbin and Bouaziz, 2009] Mathieu Barbin and Mehdi Bouaziz. A Unified Library of Primitives for QML-Compilers and Interpreters. Technical report, MLstate.

[Benayoun et al., 2009] Vincent Benayoun, David Teller, and Mikolaj Konarski. The QML type system. Technical report, MLstate. Internal document.

[Bouaziz, 2008] Mehdi Bouaziz. Approximation statique de propriétés d'applications *web*. Technical report, MLstate. Internal document.

[Cardellini et al., 1999] Valeria Cardellini, Roma I, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-server Systems. *IEEE Internet Computing*, 3:28–39.

[Chong et al., 2007] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP '07*, pages 31–44. ACM Press.

[Cooper et al., 2006] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag.

[Cousot, 1997] Patrick Cousot. Types as Abstract Interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France. ACM Press, New York, NY.

[Dargaye and Leroy, 2009] Zaynah Dargaye and Xavier Leroy. A verified framework for higher-order uncurrying optimizations. Submitted.

[Garrett, 2005] Jesse James Garrett. Ajax: A New Approach to Web Applications.

[Gold and Harman, 2007] Nicolas Gold and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM*, 16.

[Harman and Hierons, 2001] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92.

[Hunt and Scott, 1999] Galen C. Hunt and Michael L. Scott. The Coign Automatic Distributed Partitioning System. In *Operating Systems Design and Implementation*, pages 187–200.

[Johnsson, 1985] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. pages 190–203.

[Kıcıman and Livshits, 2007] Emre Kıcıman and Benjamin Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *The 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. Association for Computing Machinery, Inc.

[Livshits and Kıcıman, 2008] Benjamin Livshits and Emre Kıcıman. Doloto: Code Splitting for Network-Bound Web 2.0 Applications. In *16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 350–360, New York, NY, USA. ACM.

[Mastroeni, 2008] Isabella Mastroeni. Abstract Data Dependencies and Program Slicing: from Syntax to Abstract Semantics.

[Nielsen, 1994] Jakob Nielsen. *Usability Engineering*, chapter Response Times: The Three Important Limits. Morgan Kaufmann, San Francisco.

[Reps, 1991] Thomas Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17.

[Resig, 2009] John Resig. *Secrets of the JavaScript Ninja*. Manning.

[RockStarApps, 2007] RockStarApps. Evaluate Low Level JavaScript Performance Characteristics.

[Serrano et al., 2006] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 975–985, New York, NY, USA. ACM.

[Theurer, 2006] Tenni Theurer. Performance Research, Part 1: What the 80/20 Rule Tells Us about Reducing HTTP Requests. *Yahoo! User Interface blog*.

[Tip, 1995] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189.

[Weiser, 1984] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

[Yahoo!, 2008] Yahoo! Best Practices for Speeding Up Your Web Site.

[Yang et al., 2007] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A Unified Platform for Data Driven Web Applictions with Automatic Client-Server Partitioning. Technical report, Department of Computer Science, Cornell University.

[Zyp, 2008] Kristopher William Zyp. Ajax performance analysis. *IBM Developer Works*.