Towards Reasoning for Web Applications: An Operational Semantics for Hop

Gérard Boudol
INRIA Sophia
Antipolis-Mediterranée
gbo@sophia.inria.fr

Zhengqin Luo
INRIA Sophia
Antipolis-Mediterranée
Zhengqin.Luo@sophia.inria.fr

Tamara Rezk
INRIA Sophia
Antipolis-Mediterranée
Tamara.Rezk@sophia.inria.fr

Manuel Serrano
INRIA Sophia Antipolis-Mediterranée
Manuel.Serrano@sophia.inria.fr

Abstract

We propose a small-step operational semantics to support reasoning about web applications written in the multi-tier language HOP[1]. The semantics covers both server side and client side computations, as well as their interactions, and includes creation of web services, distributed client-server communications, concurrent evaluation of service requests at server side, elaboration of HTML documents, DOM operations, evaluation of script nodes in HTML documents and actions from HTML pages at client side.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.2 [*Programming Languages*]: Language Classifications—Design languages

General Terms Languages, Theory

Keywords Web programming, Functional languages, Multi-tier languages.

1. Introduction

The web is built atop an heterogeneous set of technologies. Traditional web development environments rely on different languages for implementing specific parts of the applications. Graphical user interfaces are declared with HTML/CSS or Flash. Client-side computations are programmed with JavaScript augmented with various APIs such as the Document Object Model (DOM) API. Communications between servers and clients involve many different protocols such as HTTP for the low level communication, XmlHttpRequest for implementing remote procedure calls, and JSON for serializing data. Server sides are frequently implemented with languages such as PHP, Java, Python, or Ruby. Using so many different tools and technologies makes it difficult to develop and maintain robust applications, it also makes it difficult to understand their precise semantics.

Semantics of web applications has not been studied globally but rather components by components. In a precursor paper Queinnec has studied the interaction model of web applications based on forms submissions [17]. This work has been pursued by Graunke and his colleagues in several publications [11, 10]. Several formal semantics for JavaScript have been proposed [16, 12] excluding the semantics of the DOM that has been first studied in [8, 9]. Various formal semantics of programming languages can be used to understand behaviors of server-side code but as precise as they are, none of them can be used to understand applications as a whole as they only cover small parts of the applications.

As a response to the emergent need of simplifying the development process of web applications, multi-tier languages have been recently proposed. Examples of such languages include HOP [19], Links [4], Swift [3], and Ur [2]. Multi-tier languages usually provide a unified syntax, typically based on a mainstream programming language syntax, where web applications can be fully specified: server and client code. These languages usually also relieve the programmer from the burden of thinking about communication protocols. The HOP programming language pushes this philosophy to the

[1] This work is supported in part by the French ANR agency, grant `ANR-09--EMER-009-01`.

APLWACA '10 June 6, Toronto, Canada.
Copyright © 2010 ACM 978-1-60558-913-8/10/06. . . $10.00

extreme by addressing all aspects of web applications and totally eliminates the need of any external language in programming these applications.

HOP [19] (http://hop.inria.fr) is based on the Scheme programming language [13] which it extends in several directions. It is multi-threaded. It provides many libraries used for implementing modern applications (mail, multimedia, ...). It also extends Scheme with constructs and APIs dedicated to web programming. The new constructs are: *i)* service definitions that are server functions associated with URLs that can be invoked by clients, *ii)* service invocations that let clients invoke servers' services, *iii)* client-side expressions that are used by servers to create client-side programs, and, *iv)* server-side expressions, that are embedded inside client-side expressions. The new APIs are: full HTML and DOM support that let servers and clients define and modify HTML documents and HTML fragments.

When a HOP program is loaded into a HOP *broker* [18], *i.e.*, the HOP execution environment, it is split and compiled on-the-fly. Sever-side parts are compiled to a mix of byte-code or native code and client-side parts are compiled to JavaScript [15]. In the source code, a syntactic mark instructs the compiler about the location where the expression is to be evaluated. Figure 1 illustrates the dual compilation.

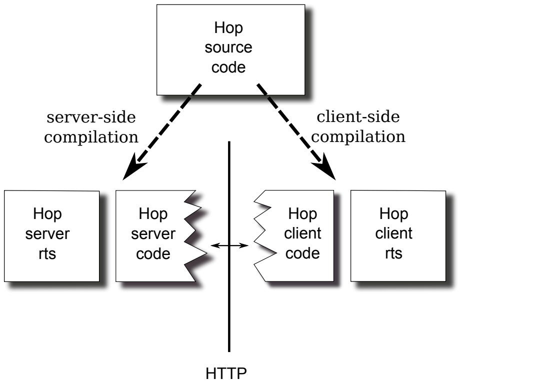

Figure 1. Hop architecture

When the HOP broker starts, it registers all the available programs and waits for client connections. Upon connection, it actually loads the program needed to fulfill the request it has received and returns a HTML document which contains the client-side part of the program to the client that has emitted the request. That client proceeds with the execution of the program. When needed, the client may invoke server-side *services* which accept client-side values and returns server-side values. The normal execution of the HOP program keeps flowing from the client to the server and vice-versa.

By covering all aspects of programming web applications, HOP can then be used to reason *globally* about these applications. Our contribution in this paper is to provide a formal and unified small-step operational semantics that could support such reasoning. A denotational continuation-based semantics was previously given for a core subset of HOP [20]. However, this work did not cope with DOM operations nor multiple clients. It only described the elaboration of client-side code as generated by the server-side code. The semantics given in this paper, in addition to being written in the more versatile style of operational semantics, covers a much wider spectrum of the language. Indeed, our semantics can support *global* formal reasoning about web applications.

For a part the HOP language, being based on Scheme, relies on standard programming constructs. However, several features of HOP are specific of a multi-tier language, and therefore require specific semantics that, as far as we can see, have not been previously formalized. These features are mostly related to the stratified design of server codes. In particular, the dynamic client code generation from the server and its installation at client site are prominent features of HOP.

Contents The paper is organized as follows: in Section 2 we describe a core of the language HOP and its semantics which is extended in Section 4 with DOM operations. In Section 3 we describe the access control mechanism in HOP. Finally, we conclude in Section 5 and propose future directions.

2. Core HOP

2.1 Syntax

In this section we introduce the syntax, and then the semantics, of the HOP language, or more precisely of the core constructs of the language. A more complete version, involving the DOM part, will be considered in Section 4. Our core language exhibits the most prominent features of the HOP language: service definition and invocation, transfer of code and values from the server to a client, that is, the distributed computing aspect of HOP. The Core HOP syntax is stratified into *server code* s and *tilde code* t. The former is basically Scheme code enriched with a construct $(\mathsf{service}\ (x)\ s)$ to define a new *service*, that is a function bound to an URL, and a construct $\tilde{}t$ to ship (tilde) code t to the client. The latter may include references $\$x$ to server values, and will be translated into *client code* c, before being shipped to the client. In the actual HOP system, the latter is compiled into JavaScript code [15], but here we ignore the compilation phase from HOP to JavaScript, as we provide a semantics at the level of (source) client code.

The syntax is given in Figure 2, where x denotes any variable. We assume given a set $\mathcal{U}rl$, disjoint from the set of variables, of names denoting URLs. These names are values in the Core HOP language, where they are used as denoting a function, or more accurately a service. When provided with an argument, that is a value w, a call $(u\,w)$ to a service is transformed into a value $u?w$ that can be passed around as an argument. In particular, such an argument will be used in the $(\mathsf{with\text{-}hop}\ u?w\ w')$ form, which sends the value w to the

$$
\begin{array}{rcll}
u & \in & \mathit{Url} & \mathit{URL} \\
s & ::= & x \mid w \mid (s_0\, s_1) \mid {}^{\sim}t & \textit{server code} \\
 & \mid & (\mathsf{service}\ (x)\ s) \mid (\mathsf{with\text{-}hop}\ s\ s) & \\
t & ::= & x \mid u \mid u?v \mid (\mathsf{lambda}\ (x)\ t) & \\
 & \mid & (t_0\, t_1) \mid \$x \mid (\mathsf{with\text{-}hop}\ t_0\ t_1) & \\
w & ::= & u \mid u?w \mid (\mathsf{lambda}\ (x)\ s) & \textit{server values} \\
 & \mid & {}^{\sim}c \mid () & \\
c & ::= & x \mid v \mid (c_0\, c_1) & \textit{client code} \\
 & \mid & (\mathsf{with\text{-}hop}\ c_0\ c_1) & \\
v & ::= & u \mid u?v \mid (\mathsf{lambda}\ (x)\ c) \mid () & \textit{client values}
\end{array}
$$

Figure 2. Core HOP Syntax

service at u somewhere in the web, and waits for a value to be returned as an argument to the continuation w'.

A server expression usually contains subexpressions of the form ${}^{\sim}t$. As we said, t represents code that will be executed at client site. This code cannot create a service, that is, it does not contain any subexpression $(\mathsf{service}\ (x)\ s)$, but it usually calls services from the server, by means of the $(\mathsf{with\text{-}hop}\ t_0\ t_1)$ construct, and it may use values provided by the server, by means of subexpressions $\$x$. When the latter are absent (that is, when they have been replaced by the value bound to x), a t expression reduces to a client expression c. Notice that for the server an expression ${}^{\sim}c$ is a value, meaning that the code c is frozen and will only be executed at client site. Values also include $()$, which is a shorthand for the *unspecified* Scheme runtime value. In a more complete description of the language we would include other kinds of values, like for instance boolean truth values, integers, strings, and so on, as well as some constructs to build and use these values. It is also easy to add Scheme imperative constructs like $(\mathsf{set!}\ x\ s)$.

As usual $(\mathsf{lambda}\ (x)\ s)$ binds x in the expression s, and the same holds for $(\mathsf{service}\ (x)\ s)$. A HOP *program* is a *closed* expression s, meaning that it does not contain any free variable (but it may contain names u for services that are provided from outside of the program). We shall consider expressions up to α-conversion, that is up to the renaming of bound variables, and we denote by $s\{y/x\}$ the expression resulting from substituting the variable y for x in s, possibly renaming y in subexpressions where this variable is bound, to avoid captures.

The operational semantics of the language will be described as a transition system, where at each step a (possibly distributed) *redex* is reduced. As usual, this occurs in specific positions in the code, that are described by means of *evaluation contexts* [5]. In order to describe in a simple way the communications between a client, invoking a service, and the server, which computes the answer to the service request, we shall introduce a new form into the syntax, namely

$$s ::= \cdots \mid (j\ s)$$

where j is a "*communication identifier*" (or channel), taken from some infinite set, disjoint from the set of variables and the set Url of URLs. The syntax of evaluation contexts is as follows:

$$
\begin{array}{rcl}
\mathbf{E} & ::= & [] \mid (\mathbf{E}\, s) \mid (w\, \mathbf{E}) \mid (j\, \mathbf{E}) \\
 & \mid & (\mathsf{with\text{-}hop}\ \mathbf{E}\ s) \mid (\mathsf{with\text{-}hop}\ w\ \mathbf{E})
\end{array}
$$

Since client code and client values are particular cases of server code and server values respectively, evaluation contexts in client code are particular cases of (server) evaluation contexts. One can see that for the $(\mathsf{with\text{-}hop}\ s_0\ s_1)$ form, one has to evaluate s_0, and then s_1, before actually calling a service. As usual, we denote by $\mathbf{E}[s]$ the result of filling the hole $[]$ in context $\mathbf{E}$ with expression s.

2.2 Semantics

The semantics of a HOP program is represented as a sequence of transitions between configurations. A configuration consists in

- a server configuration S, together with an environment μ providing the values for the variables occurring in the server configuration. The server configuration consists in a main thread executing server's code, and a number of threads of the form $(j\, s)$ executing client's requests to services.
- a client configuration C, which consists in a multiset of running clients. Each client is a tuple $\langle c, \mu, W\rangle$ where c is the running client code, typically performing service requests, μ is the local environment for the client (distinct from the one of the server: the client and the server do not share any state), and W is a multiset of pending continuations $(v\, j)$, waiting for a value returned from a service call (which has been named j), or callbacks $(v\, c)$, and more generally client code c waiting for being processed at client site (we shall see another instance of this in Section 4 with the onclick construct).
- a global environment ρ, binding URLs to the services they denote.
- a set J of communication identifiers that are currently in use.

Then a configuration Γ has the form $((S, \mu), C, \rho, J)$. However, to simplify a little the semantic rules, and to represent the concurrent execution of the various components, we shall use the following syntax for configurations:

$$\Gamma ::= \mu \mid \rho \mid J \mid s \mid \langle c, \mu, W\rangle \mid (\Gamma \parallel \Gamma)$$

where μ is a mapping from a finite set $\mathsf{dom}(\mu)$ of variables to values (server values or client values), ρ is a mapping from a finite set $\mathsf{dom}(\rho)$ of URLs to services, that is

$$
\begin{array}{rcl}
\dagger w & = & \begin{cases} \bot & \text{if } w = (\mathsf{lambda}\ (x)\ s) \\ c & \text{if } w = \tilde{}c \\ w & \text{otherwise} \end{cases} \\
\Xi(\mu, x) & = & x \\
\Xi(\mu, u) & = & u \\
\Xi(\mu, u?v) & = & u?v \\
\Xi(\mu, (\mathsf{lambda}\ (x)\ t)) & = & (\mathsf{lambda}\ (y)\ \Xi(\mu, t\{y/x\})) \\
 & & \text{where } y \notin \mathsf{dom}(\mu) \\
\Xi(\mu, (t_0 t_1)) & = & (\Xi(\mu, t_0)\Xi(\mu, t_1)) \\
\Xi(\mu, \$x) & = & \begin{cases} \bot & \text{if } \mu(x) = (\mathsf{lambda}\ (y)\ s) \\ & \text{or } \mu(x) = \tilde{}c \\ \mu(x) & \text{otherwise} \end{cases} \\
\Xi(\mu, (\mathsf{with\text{-}hop}\ t_0\ t_1)) & = & (\mathsf{with\text{-}hop}\ \Xi(\mu, t_0)\ \Xi(\mu, t_1))
\end{array}
$$

Figure 3. Server to Client

functions $(\mathsf{lambda}\ (x)\ s)$, and W is a finite set of expressions of the form $(v\,j)$[2] or $(v\,c)$. A configuration is *well-formed* if it contains exactly one μ, one ρ and one J[3]. We only consider well-formed configurations in what follows. We assume that parallel composition || is commutative and associative, so that the rules can be expressed following the "chemical style" of [1], specifying local "reactions" of the form $\Gamma \to \Gamma'$ that can take place anywhere in the configuration. That is, we have a general rule

$$\frac{\Gamma \to \Gamma'}{(\Gamma \parallel \Gamma'') \to (\Gamma' \parallel \Gamma'')}$$

meaning that if the components of Γ are present in the configuration, which can therefore be written $(\Gamma \parallel \Gamma'')$, and if these components interact to produce Γ', then we can replace the components of Γ with those of Γ'.

Before introducing and commenting the reaction rules, we need to define an auxiliary function transforming tilde code into client code. As we said a subexpression $\tilde{}t$ in server code is *not* evaluated at server side, but will be shipped to the client, usually as the answer to a service request. Since the expression t may contain references $\$x$ to server values, to define the semantics we introduce an auxiliary function Ξ that takes as arguments an environment μ and an expression t, and transforms it into a client expression c. This is defined in Figure 3, where we also introduce a partial function † that transforms a (server) value into a client expression by removing the tilde, provided the value is not a λ-abstraction. The Ξ transformation consists in replacing $\$x$ by the value bound to x in μ, but one should notice that a function, that is a $(\mathsf{lambda}\ (x)\ s)$, or client code c cannot be sent to the client this way, because this would in general result in breaking the bindings of free variables that may occur in such an expression. Then this has to be considered as an error.

The semantics is given in Figure 4, which we now comment. First notice that we write a compound configuration $(\Gamma \parallel \Gamma')$ as $\Gamma \parallel \Gamma'$. This is not ambiguous, since parallel composition is commutative and associative. When we have to evaluate a variable (rule VARS), we need to lookup into the corresponding environment μ, which we express as a reaction from $\mathbf{E}[x] \parallel \mu$, but obviously the environment must remain unchanged as a component of the configuration, which is why we restore it in $\mathbf{E}[w] \parallel \mu$, where $w = \mu(x)$. As we said when introducing the syntax, a call $(u\,w)$ to a service is transformed into a value $u?w$ (rules REQS and REQC). In server code, a subexpression $\tilde{}t$ is translated (rule TILDE) into a server value $\tilde{}c$ by means of the transformation Ξ. Evaluating $(\mathsf{service}\ (x)\ s)$ (rule SERVDEF) creates a new URL name[4] $u \notin \mathsf{dom}(\rho)$, returns this name to the evaluation context, and updates the service environment ρ by adding a new service (= function) associated with u. We may have service invocations from the server, that is $(\mathsf{with\text{-}hop}\ u?w_0\ w_1)$, where the name u refers to some pre-existing service, that has not been created by the running program. In that case (rule SERVINVOCS), we assume that the returned value w is provided by an "oracle" Ω. This value is passed as an argument to the continuation w_1. This oracle represents a call to an external service available in the web, and allows for writing mashups using HOP. Observe that service invocation from the server behaves like a RPC, whereas service invocation from a client is asynchronous: evaluating a $(\mathsf{with\text{-}hop}\ u?v_0\ v_1)$ from client side (rule SERVINVOCC) creates a new communication name j, spawns a thread $(j\ (w\,v_0))$ at server side to evaluate the request to the service, and terminates the invocation at client side while adding a continuation $(v_1\,j)$ that waits for the value returned from the server. This returned value is transformed into server code (or value) by means of the † function, and then provided as an argument to the continuation v_1 (rule SERVRET); the communication identifier j is then recovered. Concluding the semantics of the with-hop construct, a callback $(v'\ c)$ from the set W is evaluated when the client's code has terminated (rule CALLBACK). Finally, we have a last rule INIT, similar to SERVINVOCC, that models the situation where a new client shows up and sends a service request to the server, initiating a new thread of computation at server side. However in this rule the client's continuation has a special form setdoc, the meaning of which will become clear in Section 4, where this continuation is used to set up a HTML page at client side. In Core HOP, we can consider setdoc as being simply the identity $(\mathsf{lambda}\ (x)\ x)$.

Let us illustrate this semantics with an example, where we use the form $(\mathsf{let}\ ((x\,s_0))\ s_1)$ as an abbreviation for

[2] These expressions $(v\,j)$ do not evaluate, and therefore we do not need to add them to the syntax of client code.

[3] These components are omitted whenever they are empty.

[4] In the HOP language this is an optional argument to a service definition.

$$\frac{\mu(x) = w}{\mathbf{E}[x] \parallel \mu \to \mathbf{E}[w] \parallel \mu}\ (\textsc{VarS}) \qquad \frac{\mu(x) = v}{\langle \mathbf{E}[x], \mu, W\rangle \to \langle \mathbf{E}[v], \mu, W\rangle}\ (\textsc{VarC})$$

$$\frac{}{\mathbf{E}[(u\,w)] \to \mathbf{E}[u?w]}\ (\textsc{ReqS}) \qquad \frac{}{\langle \mathbf{E}[(u\,v)], \mu, W\rangle \to \langle \mathbf{E}[u?v], \mu, W\rangle}\ (\textsc{ReqC})$$

$$\frac{y \notin \mathsf{dom}(\mu)}{\mathbf{E}[((\mathsf{lambda}\ (x)\ s)w)] \parallel \mu \to \mathbf{E}[s\{y/x\}] \parallel \mu \cup \{y \mapsto w\}}\ (\textsc{AppS})$$

$$\frac{y \notin \mathsf{dom}(\mu)}{\langle \mathbf{E}[((\mathsf{lambda}\ (x)\ c)v)], \mu, W\rangle \to \langle \mathbf{E}[c\{y/x\}], \mu \cup \{y \mapsto v\}, W\rangle}\ (\textsc{AppC})$$

$$\frac{\Xi(\mu, t) = c}{\mathbf{E}[\tilde{}t] \parallel \mu \to \mathbf{E}[\tilde{}c] \parallel \mu}\ (\textsc{Tilde}) \qquad \frac{u \notin \mathsf{dom}(\rho)}{\mathbf{E}[(\mathsf{service}\ (x)\ s)] \parallel \rho \to \mathbf{E}[u] \parallel \rho \cup \{u \mapsto (\mathsf{lambda}\ (x)\ s)\}}\ (\textsc{ServDef})$$

$$\frac{u \notin \mathsf{dom}(\rho) \qquad \Omega(u, w_0) = w}{\mathbf{E}[(\mathsf{with\text{-}hop}\ u?w_0\ w_1)] \parallel \rho \to \mathbf{E}[(w_1\,w)] \parallel \rho}\ (\textsc{ServInvocS})$$

$$\frac{j \notin J \qquad \rho(u) = w}{\langle \mathbf{E}[(\mathsf{with\text{-}hop}\ u?v_0\ v_1)], \mu, W\rangle \parallel \rho \parallel J \to (j\ (w\,v_0)) \parallel \langle \mathbf{E}[()], \mu, W \cup \{(v_1\,j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{ServInvocC})$$

$$\frac{}{(j\,w) \parallel \langle c, \mu, W \cup \{(v\,j)\}\rangle \parallel J \to \langle c, \mu, W \cup \{(v\,(\dagger w))\}\rangle \parallel J - \{j\}}\ (\textsc{ServRet})$$

$$\frac{}{\langle v, \mu, \{c\} \cup W\rangle \to \langle c, \mu, W\rangle}\ (\textsc{Callback})$$

$$\frac{j \notin J \qquad \rho(u) = w}{\rho \parallel J \to (j\ (w\,v)) \parallel \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{Init})$$

Figure 4. Core HOP Semantics

$$\begin{array}{lll}
\rho_0 \ \to & (j\ ((\mathsf{lambda}\ (z)\ s)\ ())) & (\textsc{Init}) \\
 & \parallel \langle (), \emptyset, ((\mathsf{lambda}\ (x)\ x)\ j)\rangle \parallel \rho_0 \parallel \{j\} & \\
\xrightarrow{*} & (j, (\mathsf{let}\ ((x\,u_1))\ \tilde{}t)) \parallel \mu_0 & (\textsc{AppS}, \textsc{ServDef}) \\
 & \parallel \langle (), \emptyset, ((\mathsf{lambda}\ (x)\ x)\ j)\rangle \parallel \rho_1 \parallel \{j\} & \\
 & \text{where } \mu_0 = \{z' \mapsto ()\} & \\
 & \qquad \rho_1 = \rho_0 \cup \{u_1 \mapsto (\mathsf{lambda}\ (y)\ y)\} & \\
\xrightarrow{*} & (j\ \tilde{}c) \parallel \mu_1 & (\textsc{AppS}, \textsc{Tilde}) \\
 & \parallel \langle (), \emptyset, ((\mathsf{lambda}\ (x)\ x)\ j)\rangle \parallel \rho_1 \parallel \{j\} & \\
 & \text{where } c = (\mathsf{with\text{-}hop}\ (u_1\ ())\ (\mathsf{lambda}\ (x)\ x)) & \\
 & \qquad \mu_1 = \mu_0 \cup \{x' \mapsto u_1\} & \\
\xrightarrow{*} & \mu_1 \parallel \langle ((\mathsf{lambda}\ (x)\ x)\ c), \emptyset, \emptyset\rangle \parallel \rho_1 & (\textsc{ServRet}, \textsc{Callback}) \\
\xrightarrow{*} & (j\ ((\mathsf{lambda}\ (y)\ y)\ ())) \parallel \mu_1 & (\textsc{ReqC}, \textsc{ServInvocC}) \\
 & \parallel \langle ((\mathsf{lambda}\ (x)\ x)\ ()), \emptyset, ((\mathsf{lambda}\ (x)\ x)\ j)\rangle \parallel \rho_1 \parallel \{j\} & \\
\xrightarrow{*} & (j\ ()) \parallel \mu_2 & (\textsc{AppS}, \textsc{VarS}, \textsc{AppC}, \textsc{VarC}) \\
 & \parallel \langle (), \mu_0', ((\mathsf{lambda}\ (x)\ x)\ j)\rangle \parallel \rho_1 \parallel \{j\} & \\
 & \text{where } \mu_2 = \mu_1 \cup \{y' \mapsto ()\} & \\
 & \qquad \mu_0' = \{x' \mapsto ()\} & \\
\xrightarrow{*} & \mu_2 \parallel \langle (), \mu_1', \emptyset\rangle \parallel \rho_1 & (\textsc{ServRet}, \textsc{Callback}, \textsc{AppC}, \textsc{VarC}) \\
 & \text{where } \mu_1' = \mu_0' \cup \{z \mapsto ()\} &
\end{array}$$

Figure 5. Operational Semantics: an Example

((lambda (x) s_1) s_0). Let

$$\begin{array}{rcl} s & = & (\mathsf{let}\ ((x\ (\mathsf{service}\ (y)\ y)))\ \tilde{\ }t) \\ t & = & (\mathsf{with\text{-}hop}\ (\$x\ ())\ (\mathsf{lambda}\ (x)\ x)) \end{array}$$

We start with a configuration where there is a service (lambda (z) s) available at URL u_0, that is with $\rho_0 = \{u_0 \mapsto (\mathsf{lambda}\ (z)\ s)\}$ (and $\mu = \emptyset = J$, so we omit these components). Then we have the transitions shown in Figure 5, which displays service definition and the interactions between clients and a server. In the first step we interpret the setdoc continuation as the identity, as explained above. One can see (in the last steps) that server threads compute concurrently with clients. However, one should observe that, since the server and the clients do not share any common state, there is no conflict between the server and client computations, nor among client computations. This means that, when reasoning about the behavior of a HOP program, we do not have to consider all the possible interleavings, since many steps are actually independent from each other. In fact, we could have presented the semantics using a synchronous style, where a client always waits for the answers from the server before resuming its own computations. That is, we could have restricted the VARC, REQC, APPC and SERVINVOCC rules to the case where the set W only contains callbacks of the form $(v\,c)$, and no pending continuation $(v\,j)$. This is not the way a HOP program actually behaves, but this restriction to the semantics does not change it in an essential manner, if the services always return. In any case, one should be able to use local reasoning for server and client code.

2.3 Implementation

HOP rests on traditional web technologies to execute programs. The server side code is executed by a bootstrapped web server [18] that embeds an on-the-fly compiler that compiles client-side code to JavaScript [15]. Communications between servers and clients implement the HTTP protocol [6].

The actual service definition construct (service) let servers associate arbitrary URLs to services. These values, represented by $u?v$ in the semantics (see Figure 2), are used in web browsers URL bars to spawn HOP executions. For instance, provided with a server accepting connections on port 8080, pointing a browser to the URL

`http://localhost:8080/hop/doc`

forces the server to load the interactive documentation program and to start its execution. This delivers a web page implementing the GUI of the live documentation. This action is modeled by the INIT rule of the semantics (see Figure 4).

The mapping from URLs to services, named μ in the semantics rules such as the SERVDEF in Figure 4, is implemented on the server by a global hash table which is not automatically garbage collected. When a service is defined, the program may add annotations such as *time-to-live* or *timeout* that enables the server to remove expired services from the environment.

$$s \quad ::= \quad \ldots \mid (\mathsf{sservice}\ R\ (x)\ s) \qquad \textit{server extension}$$

Figure 6. Access Control Extension for HOP Syntax

The implementation of service invocation (the with-hop construct, the semantics of which is given by rules SERVINVOCS, SERVINVOCC, SERVRET, and CALLBACK) is implemented with the famous `XmlHttpRequest` object of *Ajax* applications. Upon service invocation, the actual arguments are marshalled using a format suitable for URLs (rule SERVINVOCC). On return, values are marshalled by the server using the JSON convention. Using that particular encoding enables fast browser unmarshaling.

3. Access Control

As an option to the interaction between clients and server, HOP proposes access control facilities. In this section we show how these facilities can be formally modeled. Access control permissions in HOP include service execution permission and read permission for directories. For simplicity, we only model service execution permissions here. We introduce the set $\mathcal{C}redentials$ of credentials which is disjoint from $\mathcal{U}rl$ and variables. A credential $cr \in \mathcal{C}redentials$ can be seen as an abstract representation of username and password, which serves as a token for access. We use R for denoting a set of credentials. To model services protected by a simple access control protocol, we assume a specific form (sservice R (x) s) of defining a "secured service." The HOP syntax is extended as shown in 6. When defining a service protected by access control, one needs to provide a set of credentials, such that only those who hold one of the credentials can access the service. Accordingly, we extend the definition for ρ and client configuration:

- The global environment ρ binds urls to a pair (R, w) for some R and w. The set R denotes permitted access rights for the service;
- The client configuration C now is a tuple $\langle cr, c, \mu, W\rangle$, where cr is the credential held by the client.

We update the semantics rules for access control in Figure 7. Only rules concerning access control are shown. For the other rules, cr in a client configuration is left unchanged. When defining a new service, rule SERVDEF binds the url with a set of credential provided and a service body. The INIT rule models that a new client with a credential sends a service request. The request takes place only when the credential is in the set of credentials related to the requested service. The SERVINVOCC rule, similar to INIT, also checks the credential.

$$\frac{u \notin \mathsf{dom}(\rho)}{\mathbf{E}[(\mathsf{sservice}\ R\ (x)\ s)] \parallel \rho \to \mathbf{E}[u] \parallel \rho \cup \{u \mapsto (R, (\mathsf{lambda}\ (x)\ s))\}}\ (\textsc{ServDef})$$

$$\frac{j \notin J \qquad \rho(u) = (R, w) \qquad cr \in R}{\langle cr, \mathbf{E}[(\mathsf{with\text{-}hop}\ u?v_0\ v_1)], \mu, W\rangle \parallel \rho \parallel J \to (j\ (w\ v_0)) \parallel \langle cr, \mathbf{E}[()], \mu, W \cup \{(v_1\ j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{ServInvocC})$$

$$\frac{j \notin J \qquad \rho(u) = (R, w) \qquad cr \in R}{\rho \parallel J \to (j\ (w\ v_0)) \parallel \langle cr, (), \emptyset, \{(v_1\ j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{Init})$$

Figure 7. Core HOP Extended with Access Control

HOP implementation of access control relies on HTTP [6] which accommodates authentications with two different schemas known as *Basic* and *Digest Access Authentication* [7]. Basic authentication sends user identity and password as plain texts in the header of the request. Digest authentication sends the user identity and password encrypted with a nonce. Digest authentication is therefore slightly more secure than Basic authentication. A third traditional means for authenticating requests consists in including user identity and password in the host part of the URLs. This method which offers roughly the same (in)security as the Basic authentication scheme is widely used because it eases HTTP requests to be scripted.

HOP rests on these standard mechanisms to authenticate requests. Upon reception, requests are authenticated by parsing the header fields. A special *anonymous* user is used as a default authentication for requests that contain no valid authentication information. Users are declared in HOP configuration files. They are created at the very beginning of the server execution, before the first request is received. An user is a data structure containing a name for identification, an encrypted password, a list of files and directories it is allowed to access to, a list of services it is allowed to execute, and a list of groups it belongs to. When a service is about to be executed in response to a client call, HOP automatically authenticates the request and checks if that user is granted the permission to execute the service. When authorized, the execution proceeds. Otherwise, HOP returns a dedicated HTTP response code (401) that forces the client browser to pops up a login window and resubmit the request with a different authentication. In addition to the automatic authentication, HOP programs may also explicitly check user permissions. Hence, any service may enforce it's own security policy.

$$\begin{array}{rcl}
p,\ q,\ r\ldots & \in & \mathcal{P}\mathit{ointer} \\
s & ::= & \cdots \mid (\langle tag\rangle\ s) \mid (\langle tag\rangle\ :\mathsf{onclick}\, s_0\ s_1) \\
 & \mid & (\mathsf{dom\text{-}append\text{-}child!}\ s_0\ s_1) \\
t & ::= & \cdots \mid (\langle tag\rangle\ t) \mid (\langle tag\rangle\ :\mathsf{onclick}\, t_0\ t_1) \\
 & \mid & (\mathsf{dom\text{-}append\text{-}child!}\ t_0\ t_1) \\
w & ::= & \cdots \mid p \\
c & ::= & \cdots \mid (\langle tag\rangle\ c) \mid (\langle tag\rangle\ :\mathsf{onclick}\, c_0\ c_1) \\
 & \mid & (\mathsf{dom\text{-}append\text{-}child!}\ c_0\ c_1) \\
v & ::= & \cdots \mid p \\
tag & ::= & \mathsf{HTML} \mid \mathsf{DIV} \mid \cdots
\end{array}$$

Figure 8. DOM Extension for HOP Syntax

4. DOM Extension

In Section 2 we have seen how distributed computations are built and run in HOP, but apart from that we have seen no interesting effect of a HOP program. In this section we consider another part of the HOP language, which allows one to built HTML trees, that will be interpreted and displayed by the client's browser. The client can manipulate the host HTML page by means of the DOM (Document Object Model [14]) interface of the browser. Then we enrich the syntax with some basic HTML constructs, written in Scheme style, and operations supported by the DOM. Here we content ourselves to consider the HTML and DIV tags, and the $(\mathsf{dom\text{-}append\text{-}child!}\ s_0\ s_1)$ construct – the other ones are similar (see [8]). The HOP syntax is extended as shown in Figure 8, where we assume given an infinite set $\mathcal{P}\mathit{ointer}$ of *pointers*, that will be used to denote nodes in HTML trees. The pointers are run-time values. Notice that instead of writing $\langle tag\rangle \cdots \langle /tag\rangle$ as in HTML, we write $(\langle tag\rangle\ \cdots)$ in HOP, which means that a tag is a function that is used to build an HTML document. The general form in HOP is

$$(\langle tag\rangle\,[:\ attr]\ s)$$

where *attr* is an optional list of attributes. We only consider here the cases where there is no attribute, or where this attribute is $\mathsf{onclick}\, s$, but we sometimes write $(\langle tag\rangle\,[:\ attr]\ s)$ for any of the two forms, when the attribute is irrelevant. The optional $\mathsf{onclick}\, s$ attribute offers to the client the possibility of running some code (namely c if $s = \tilde{}c$), by clicking on the node. (In HOP there are other similar facilities.)

The semantics of the $(\langle tag\rangle\,[:\ attr]\ \cdots)$ construct is that it builds a node of a tree in a *forest*. In order to define this, we assume given a specific null pointer, denoted α, which is not in $\mathcal{P}\mathit{ointer}$. We use π to range over $\mathcal{P}\mathit{ointer} \cup$

$$\frac{\mu(r) = (\alpha, (\langle\mathsf{HTML}\rangle\,[:\,attr]\;\ell)}{(j\,r)\parallel\mu\parallel\langle(),\emptyset,\{(\mathsf{setdoc}\,j)\},\alpha\rangle\parallel J\to\mu\parallel\langle r,\mu\lceil r,\emptyset,r\rangle\parallel J-\{j\}}\;(\textsc{ServRet1})$$

$$\frac{\mathsf{dom}(\mu_0\lceil w)\cap\mathsf{dom}(\mu_1)=\emptyset}{(j\,w)\parallel\mu_0\parallel\langle c,\mu_1,W\cup\{(v\,j)\},r\rangle\parallel J\to\langle c,\mu_1[\mu_0\lceil w],W\cup\{(v\,(\dagger w))\},r\rangle\parallel J-\{j\}}\;(\textsc{ServRet2})$$

Figure 9. HOP Semantics (Modified Rules)

$\{\alpha\}$. Then a forest maps (non null) pointers to pairs made of a (possibly null) pointer and an expression of the form $(\langle tag\rangle\,[:\;attr]\;c_1+\cdots+c_n)$. The pointer $q\in\mathcal{Pointer}$ assigned to p is the *ancestor* of the node, if it exists. If it does not, this pointer is α. Such a node is labelled tag and has n children, which are either leafs (labelled with some client code or value) or pointers to other nodes in the tree. For simplicity we consider here the forest as joined to the environment providing values for variables. That is, we now consider that μ is a mapping from a set $\mathsf{dom}(\mu)$ of variables and (non null) pointers, that maps variables to values, and pointers to pairs made of a (possibly null) pointer and a *node* expression. The syntax for node expressions a is as follows:

$$\begin{array}{rcl} a & ::= & (\langle tag\rangle\;\ell)\;\mid\;(\langle tag\rangle\;:\mathsf{onclick}\,c\;\ell)\\ \ell & ::= & \varepsilon\;\mid\;c\;\mid\;(\ell_0+\ell_1)\end{array}$$

where ε is the empty list. In what follows we assume that $+$ is associative, and that $\varepsilon+\ell=\ell=\ell+\varepsilon$. We shall also use the following notations in defining the semantics, assuming that the pointers occuring in the list ℓ are distinct:

$$\begin{array}{rcl} (\langle tag\rangle\,[:\,attr]\;\ell)+p & = & (\langle tag\rangle\,[:\,attr]\;\ell+p)\\ (\langle tag\rangle\,[:\,attr]\;\ell_0+p+\ell_1)-p & = & (\langle tag\rangle\,[:\,attr]\;\ell_0+\ell_1)\end{array}$$

Given a forest μ, and $p\in\mathsf{dom}(\mu)$, we denote by $\mu[p\mapsto(\pi,a)]$ the forest obtained by updating the value associated with p in μ. More generally, we define $\mu[\mu']$ as follows:

$$\begin{array}{rcl} \mathsf{dom}(\mu[\mu']) & = & \mathsf{dom}(\mu)\cup\mathsf{dom}(\mu')\\ (\mu[\mu'])(p) & = & \begin{cases}\mu'(p) & \text{if } p\in\mathsf{dom}(\mu')\\ \mu(p) & \text{otherwise}\end{cases}\end{array}$$

For $P\subseteq\mathsf{dom}(\mu)$, we also define $\mu\lceil P$ to be the least subset of μ satisfying

$$\begin{array}{l} I\subseteq\mathsf{dom}(\mu\lceil P)\\ q\in\mathsf{dom}(\mu\lceil P)\;\&\;\mu(q)=(q',(\langle tag\rangle\,[:\,attr]\;\ell)\\ \quad\Rightarrow\;q'\in\mathsf{dom}(\mu\lceil P)\\ q\in\mathsf{dom}(\mu\lceil P)\;\&\;\mu(q)=(\pi,(\langle tag\rangle\,[:\,attr]\;\ell_0+q'+\ell_1)\\ \quad\Rightarrow\;q'\in\mathsf{dom}(\mu\lceil P)\\ q\in\mathsf{dom}(\mu\lceil P)\;\Rightarrow\;(\mu\lceil P)(q)=\mu(q)\end{array}$$

We overload this notation by writing $\mu\lceil c$ for $\mu\lceil P$ where P is the set of pointers that occur in c. This is the forest that is the part of μ relevant for the expression c.

The syntax of evaluation contexts needs to be extended, but we also have now to distinguish client's evaluation contexts $\mathbf{C}$ from server's evaluation contexts $\mathbf{S}$:

$$\begin{array}{rcl} \mathbf{S} & ::= & \cdots\mid(\langle tag\rangle\,\mathbf{S})\mid(\langle tag\rangle\;:\mathsf{onclick}\,\mathbf{S}\;s)\\ & \mid & (\langle tag\rangle\;:\mathsf{onclick}\,w\;\mathbf{S})\\ & \mid & (\mathsf{dom\text{-}append\text{-}child!}\;\mathbf{S}\;s)\\ & \mid & (\mathsf{dom\text{-}append\text{-}child!}\;w\;\mathbf{S})\\ \mathbf{C} & ::= & \cdots\mid(\langle tag\rangle\,\mathbf{C})\mid(\langle tag\rangle\;:\mathsf{onclick}\,c\;\mathbf{C})\\ & \mid & (\mathsf{dom\text{-}append\text{-}child!}\;\mathbf{C}\;c)\\ & \mid & (\mathsf{dom\text{-}append\text{-}child!}\;v\;\mathbf{C})\end{array}$$

The main difference is that at client side we do not evaluate c_0 in $(\langle tag\rangle\;:\mathsf{onclick}\,c_0\;c_1)$, because this is the code that will be executed at client side when an "onclick" action is performed. Finally, as regards configurations, we now assume that clients are *rooted*. That is, a client configuration now has the form

$$\langle c,\mu,W,r\rangle$$

where the pointer r is the *root* of the HTML page that is displayed at the client site by the browser.

The semantic rules given in Figure 4 still hold, except for SERVRET, which we redefine below. We also have to extend the $\Xi(\mu,t)$ function in rule TILDE to take into account the new constructs. This is done in the obvious way, preserving the structure of the expression (the function Ξ only has an effect on the $\$x$ subexpressions), with $\Xi(\mu,p)=p$ for any $p\in\mathcal{Pointer}$. The modified rules are given in Figure 9, while the new ones are in Figure 10. The VARC, REQC, APPC, CALLBACK, SERVINVOCC and INIT rules of Figure 4 have to be adapted to suit the new form of a client configuration, which involves a root. This is done in the obvious way – so we omit to write the adapted rules –, except that in the INIT rule, the client is not yet rooted, or more precisely its root is α: the client is waiting for an HTML tree (with a root) to be provided by the server. This is formalized in rule SERVRET1: the server sends a root r, together with the associated tree $\mu\lceil r$, which should satisfy some well-formedness condition to be displayed by the browser. Here we only require that the node at r denotes an $\langle\mathsf{HTML}\rangle$ node, without any ancestor. In this rule the evaluation of $(\mathsf{setdoc}\,r)$, which is supposed to have the (here invisible) side effect of displaying something of $\mu\lceil r$, immediately returns r. The SERVRET2 rule is the same as SERVRET of Core HOP, except that some forest may also be returned, which should not conflict with the current client's HTML forest. The reader will notice the disymmetry between the rules for passing a

$$\frac{\mathsf{R}(r,p) \quad \mu(p) = (q, (\langle tag\rangle\, [:\, attr]\ \ell_0 + c + \ell_1))}{\langle v, \mu, W, r\rangle \to \langle c, \mu[p \mapsto (q, (\langle tag\rangle\, [:\, attr]\ \ell_0 + \ell_1))], W, r\rangle}\ (\textsc{Script})$$

$$\frac{\mathsf{Q}(r,p) \quad \mu(p) = (q, (\langle tag\rangle\ :\mathsf{onclick}\, c_0\ \ell))}{\langle c_1, \mu, W, r\rangle \to \langle c_1, \mu, W \cup \{c_0\}, r\rangle}\ (\textsc{Onclick})$$

$$\frac{w \notin \mathcal{P}ointer \qquad p \notin \mathsf{dom}(\mu)}{\mathbf{S}[(\langle tag\rangle\ w)] \parallel \mu \to \mathbf{S}[p] \parallel \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\ \dagger w))\}}\ (\textsc{TagVS1})$$

$$\frac{p \notin \mathsf{dom}(\mu) \qquad \mu(q) = (\alpha, a)}{\mathbf{S}[(\langle tag\rangle\ q)] \parallel \mu \to \mathbf{S}[p] \parallel \mu[q \mapsto (p, a)] \cup \{p \mapsto (\alpha, (\langle tag\rangle\ q))\}}\ (\textsc{TagIS1})$$

$$\frac{w_1 \notin \mathcal{P}ointer \qquad p \notin \mathsf{dom}(\mu)}{\mathbf{S}[(\langle tag\rangle\ :\mathsf{onclick}\, w_0\ w_1)] \parallel \mu \to \mathbf{S}[p] \parallel \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\ :\mathsf{onclick}\, \dagger w_0\ \dagger w_1))\}}\ (\textsc{TagVS2})$$

$$\frac{p \notin \mathsf{dom}(\mu) \qquad \mu(q) = (\alpha, a)}{\mathbf{S}[(\langle tag\rangle\ :\mathsf{onclick}\, w\ q)] \parallel \mu \to \mathbf{S}[p] \parallel \mu[q \mapsto (p, a)] \cup \{p \mapsto (\alpha, (\langle tag\rangle\ :\mathsf{onclick}\, \dagger w\ q))\}}\ (\textsc{TagIS2})$$

$$\frac{v \notin \mathcal{P}ointer \qquad p \notin \mathsf{dom}(\mu)}{\langle \mathbf{C}[(\langle tag\rangle\, [:\, attr]\ v)], \mu, W, r\rangle \to \langle \mathbf{C}[p], \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\, [:\, attr]\ v))\}, W, r\rangle}\ (\textsc{TagVC})$$

$$\frac{p \notin \mathsf{dom}(\mu) \qquad \mu(q) = (\alpha, b)}{\langle \mathbf{C}[(\langle tag\rangle\, [:\, attr]\ q)], \mu, W, r\rangle \to \langle \mathbf{C}[p], \mu[q \mapsto (p, b)] \cup \{p \mapsto (\alpha, (\langle tag\rangle\, [:\, attr]\ q))\}, W, r\rangle}\ (\textsc{TagIC})$$

$$\frac{\mu(p) = (\pi, a_0) \quad \mu(q) = (q', a_1) \quad \mu(q') = (\pi', a_2)}{\mathbf{S}[(\mathsf{dom\text{-}append\text{-}child!}\ p\ q)] \parallel \mu \to \mathbf{S}[()] \parallel \mu[p \mapsto (\pi, a_0 + q), q \mapsto (p, a_1), q' \mapsto (\pi', a_2 - q)]}\ (\textsc{AppendS1})$$

$$\frac{\mu(p) = (\pi, a_0) \quad \mu(q) = (\alpha, a_1)}{\mathbf{S}[(\mathsf{dom\text{-}append\text{-}child!}\ p\ q)] \parallel \mu \to \mathbf{S}[()] \parallel \mu[p \mapsto (\pi, a_0 + q), q \mapsto (p, a_1)]}\ (\textsc{AppendS2})$$

$$\frac{\mu(p) = (\pi, b_0) \quad \mu(q) = (q', b_1) \quad \mu(q') = (\pi', b_2)}{\langle \mathbf{C}[(\mathsf{dom\text{-}append\text{-}child!}\ p\ q)], \mu, W, r\rangle \to \langle \mathbf{C}[()], \mu[p \mapsto (\pi, b_0 + q), q \mapsto (p, b_1), q' \mapsto (\pi', b_2 - q)], W, r\rangle}\ (\textsc{AppendC1})$$

$$\frac{\mu(p) = (\pi, b_0) \quad \mu(q) = (\alpha, b_1)}{\langle \mathbf{C}[(\mathsf{dom\text{-}append\text{-}child!}\ p\ q)], \mu, W, r\rangle \to \langle \mathbf{C}[()], \mu[p \mapsto (p', b_0 + q), q \mapsto (p, b_1)], W, r\rangle}\ (\textsc{AppendC2})$$

Figure 10. HOP DOM Semantics

value from the server to a client (SERVRET1 & 2), which “drags” a tree with it, and for passing a value from a client to the server, as an argument for a service call (SERVINVOCC), which does not pass a tree. This is because we have found no interesting use for that. Consequently, in the current version of HOP it is an error to use a client’s node at server site.

The document sent by the server to the client upon initialization may contain some code to execute, and also opportunities for interactions from the client, which in our simplifying presentation of the HOP language only consists in $\mathsf{onclick}\, c$ expressions. Then there is a phase in which the browser, while interpreting the document sent by the server, will execute client code that is contained into the document. This is expressed by the SCRIPT rule, where the predicate $\mathsf{R}(r, p)$ means that p is a descendant of r, and that the code that we find at node p, and which is to be triggered, is the leftmost one in the tree $\mu \lceil r$ determined by r. (We should also check that this tree is still a valid HTML document. We do not formally define this predicate here – this is straightforward.) When this has been done, the client may interact with the server, by clicking on an active node. This is expressed by the rule ONCLICK, where again we have a precondition $\mathsf{Q}(r, p)$, meaning that p is a descendant from r, and that there is no code left to execute (by means of the SCRIPT rule) in the tree (again we do not formally define this predicate here).

We have already explained the next six rules, from TAGVS1 to TAGIC, that describe the construction of the server (resp. client) forest from the $(\langle tag\rangle\ s)$ and $(\langle tag\rangle\ :\mathsf{onclick}\, s_0\ s_1)$ (resp. $(\langle tag\rangle\, [:\, attr]\ c)$) expressions. We distinguish two cases for each of these constructions: if the child of the node is a value which is not a pointer, then we just create a corresponding new node in the forest. If this is a pointer q, pointing to another node in the forest, we again create a new node identified by some p, but we must also update the ancestor of q, which becomes p. In the rules for the server we see that the server values are transformed into client values or client code by means of the † function.

The remaining rules describe how the DOM operation we consider, that is $(\mathsf{dom\text{-}append\text{-}child!}\ s_0\ s_1)$ computes: first the expressions s_0 and s_1 have to be evaluated. They are supposed to return pointers p and q pointing to nodes in the forest. Then one updates the node at p, moving q as a new child of p, as well as the node at the ancestor of q (if any) which loses its q child, and we update q's ancestor to be p. It is easy to formalize the other DOM constructs in a similar way (see [8, 9]). This will be done in a more complete formal description of the semantics of the HOP language.

Let us illustrate the semantics with DOM extension with a few examples. For all the following examples, we start with a configuration where there is a service $(\mathsf{lambda}\ (z)\ s_0)$ available at URL u_0, that is with $\rho_0 = \{u_0 \mapsto (\mathsf{lambda}\ (z)\ s_0)\}$.

Example 1 This example demonstrates how DOM tree is manipulated and transmitted from server-side to client-side. Let

$$\begin{aligned}
s_0 &= (\mathsf{let}\ ((x\ (\mathsf{service}\ (y)\ (\langle\mathsf{DIV}\rangle\ y))))\ s_1)\\
s_1 &= (\mathsf{let}\ ((d\ (\langle\mathsf{DIV}\rangle\ ())))\ s_2)\\
s_2 &= (\mathsf{let}\ ((h\ (\langle\mathsf{HTML}\rangle\ d)))\ s_3)\\
s_3 &= (\mathsf{let}\ ((k\ (\mathsf{dom\text{-}append\text{-}child!}\ h\ (\langle\mathsf{DIV}\rangle\ \tilde{}t))))\ h)\\
t &= (\mathsf{with\text{-}hop}\ (\$x\ ())\\
&\qquad (\mathsf{lambda}\ (x)\ (\mathsf{dom\text{-}append\text{-}child!}\ \$d\ x)))
\end{aligned}$$

The transitions are shown in Figure 11, where the service ships a HTML tree containing a piece of client node. The client code, evaluated in client-side, requests a new tree from the server and appends it to the current document.

Example 2 This example shows how script nodes are evaluated in client-side, especially the evaluation order.

$$\begin{aligned}
s_0 &= (\mathsf{let}\ ((d\ (\langle\mathsf{DIV}\rangle\ \tilde{}t_0)))\ s_1)\\
s_2 &= (\mathsf{let}\ ((h\ (\langle\mathsf{HTML}\rangle\ d)))\ s_2)\\
s_3 &= (\mathsf{let}\ ((c\ (\mathsf{dom\text{-}append\text{-}child!}\ h\ (\langle\mathsf{DIV}\rangle\ \tilde{}t_1))))\ h)\\
t_0 &= ((\mathsf{lambda}\ (y)\ y)\ ())\\
t_1 &= ((\mathsf{lambda}\ (x)\ x)\ ())
\end{aligned}$$

We then have transitions shown in Figure 12, where transitions regarding tree construction and transmission in server-side are omitted. The tree transmitted to client contains two pieces of code. The left one c_0 will be evaluated before the right one c_1.

Example 3 This example demonstrates the ability of running code by invoking "onclick" attribute.

$$\begin{aligned}
s_0 &= (\langle\mathsf{HTML}\rangle\ (\langle\mathsf{DIV}\rangle\ :\mathsf{onclick}\ \tilde{}t))\\
t &= ((\mathsf{lambda}\ (x)\ x)\ ())
\end{aligned}$$

Here are some states in the execution of this program:

$$\begin{aligned}
\rho_0 &\to (j\ ((\mathsf{lambda}\ (z)\ s_0)\ ()))\\
&\qquad \parallel \langle (), \emptyset, (\mathsf{setdoc}\, j), \alpha\rangle \parallel \rho_0 \parallel \{j\}\\
&\xrightarrow{*} \mu_0 \parallel \langle q, \mu_1, \emptyset, q\rangle \parallel \rho_1\\
&\qquad \text{where } \mu_1 = \{q \mapsto (\alpha, (\langle\mathsf{HTML}\rangle\ p)),\\
&\qquad\qquad\qquad p \mapsto (q, (\langle\mathsf{DIV}\rangle\ :\mathsf{onclick}\ c))\}\\
&\qquad\qquad c = ((\mathsf{lambda}\ (x)\ x)\ ())\\
&\xrightarrow{*} \mu_0 \parallel \langle (), \mu_2, \emptyset, q\rangle \parallel \rho_1\\
&\qquad \text{where } \mu_2 = \mu_1 \cup \{x' \mapsto ()\}
\end{aligned}$$

Example 4 This example shows that only valid HTML document is meaningful as an answer to initial client request. Let

$$\begin{aligned}
s_0 &= (\langle\mathsf{DIV}\rangle\ \tilde{}t)\\
t &= ((\mathsf{lambda}\ (x)\ x)\ ())
\end{aligned}$$

Since the returning tree will not be a valid HTML document, the computation will be blocked by rule SERVRET1.

5. Conclusion

We have presented a small step operational semantics for the multi-tier programming language HOP, covering both the server side and client side computations, and their interactions. This semantics may therefore be used to reason *globally* about the behavior of web applications. As a benefit, language based techniques for analysis and verification are thus made available to web applications, opening new research directions to tackle security and reliability problems for the web from a programming viewpoint.

References

[1] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the ACM International Conference on Principle of Programming Languages (POPL)*, pages 81–94. ACM Press, New York, 1990.

[2] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *International Conference on Programming Languages and Implementation (PLDI)*, Toronto, Canada, June 2010.

[3] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Communications of the ACM*, 52(2):79–87, 2009.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects*, November 2006.

[5] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[6] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. Technical Report RFC 2616, The Internet Society, 1999.

$$\begin{array}{lll}
\rho_0 & \to & (j\,((\mathsf{lambda}\ (z)\ s_0)\ ())) \\
 & & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_0 \parallel \{j\}
\end{array}
\qquad (\textsc{Init})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,(\mathsf{let}\ ((x\,u_1))\ s_1)) \parallel \mu_0 \\
 & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } \mu_0 = \{z' \mapsto ()\} \\
 & \qquad\quad \rho_1 = \rho_0 \cup \{u_1 \mapsto (\mathsf{lambda}\ (y)\ (\langle\mathsf{DIV}\rangle\ y))\}
\end{array}
\qquad (\textsc{AppS}, \textsc{ServDef})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,(\mathsf{let}\ ((d\,p))\ s_2')) \parallel \mu_1 \\
 & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } s_2' = s_2\{x'/x\} \\
 & \qquad\quad \mu_1 = \mu_0 \cup \{x' \mapsto u_1, p \mapsto (\alpha, (\langle\mathsf{DIV}\rangle\ ()))\}
\end{array}
\qquad (\textsc{AppS}, \textsc{TagVS1})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,(\mathsf{let}\ ((h\,q))\ s_3')) \parallel \mu_2 \\
 & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } s_3' = s_3\{x'/x, d'/d\} \\
 & \qquad\quad \mu_2 = \mu_1[p \mapsto (q, (\langle\mathsf{DIV}\rangle\ ()))] \\
 & \qquad\qquad\qquad \cup \{d' \mapsto p, q \mapsto (\alpha, (\langle\mathsf{HTML}\rangle\ p))\}
\end{array}
\qquad (\textsc{AppS}, \textsc{VarS}, \textsc{TagIS1})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,(\mathsf{let}\ ((k\,()))\ h')) \parallel \mu_3 \\
 & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } \mu_3 = \mu_2[q \mapsto (\alpha, (\langle\mathsf{HTML}\rangle\ p\,r))] \\
 & \qquad\qquad\qquad \cup \{h' \mapsto q, r \mapsto (q, (\langle\mathsf{DIV}\rangle\ \tilde{}c))\} \\
 & \qquad\quad c = (\mathsf{with\text{-}hop}\ (u_1\ ())\ (\mathsf{lambda}\ (x)\ (\mathsf{dom\text{-}append\text{-}child!}\ p\ x)))
\end{array}
\qquad (\textsc{AppS}, \textsc{VarS}, \textsc{Tilde}, \textsc{TagVS1}, \textsc{AppendS2})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,q) \parallel \mu_4 \\
 & \parallel \langle (), \emptyset, (\mathsf{setdoc}\,j), \alpha\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } \mu_4 = \mu_3 \cup \{k' \mapsto ()\}
\end{array}
\qquad (\textsc{AppS}, \textsc{VarS})$$

$$\begin{array}{ll}
\overset{*}{\to} & \mu_4 \parallel \langle q, \mu_5, \emptyset, q\rangle \parallel \rho_1 \\
 & \text{where } \mu_5 = \{q \mapsto (\alpha, (\langle\mathsf{HTML}\rangle\ p\,r)), p \mapsto (q, (\langle\mathsf{DIV}\rangle\ ()))\} \\
 & \qquad\qquad\qquad \cup \{r \mapsto (q, (\langle\mathsf{DIV}\rangle\ c))\}
\end{array}
\qquad (\textsc{ServRet1})$$

$$\begin{array}{ll}
\overset{*}{\to} & \mu_4 \parallel \langle c, \mu_6, \emptyset, q\rangle \parallel \rho_1 \\
 & \text{where } \mu_6 = \mu_5[r \mapsto (q, (\langle\mathsf{DIV}\rangle\ \varepsilon))]
\end{array}
\qquad (\textsc{Script})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,((\mathsf{lambda}\ (y)\ (\langle\mathsf{DIV}\rangle\ y))\ ())) \parallel \mu_4 \\
 & \parallel \langle (), \mu_6, ((\mathsf{lambda}\ (x)\ (\mathsf{dom\text{-}append\text{-}child!}\ p\ x))\ j), q\rangle \parallel \rho_1 \parallel \{j\}
\end{array}
\qquad (\textsc{ReqC}, \textsc{ServInvocC})$$

$$\begin{array}{ll}
\overset{*}{\to} & (j\,r') \parallel \mu_7 \\
 & \parallel \langle (), \mu_6, ((\mathsf{lambda}\ (x)\ (\mathsf{dom\text{-}append\text{-}child!}\ p\ x))\ j), q\rangle \parallel \rho_1 \parallel \{j\} \\
 & \text{where } \mu_7 = \mu_4 \cup \{y' \mapsto (), r' \mapsto (\alpha, (\langle\mathsf{DIV}\rangle\ ()))\}
\end{array}
\qquad (\textsc{AppS}, \textsc{TagVS1})$$

$$\begin{array}{ll}
\overset{*}{\to} & \mu_7 \parallel \langle (), \mu_8, \emptyset, q\rangle \parallel \rho_1 \\
 & \text{where } \mu_8 = \mu_6[p \mapsto (q, (\langle\mathsf{DIV}\rangle\ ()\ r'))] \\
 & \qquad\qquad\qquad \cup \{r' \mapsto (p, (\langle\mathsf{DIV}\rangle\ ())), x' \mapsto r'\}
\end{array}
\qquad (\textsc{ServRet2}, \textsc{Callback}, \textsc{AppC}, \textsc{VarC}, \textsc{AppendC2})$$

Figure 11. DOM Extensions: Example 1

$$\begin{array}{lll}
\rho_0 & \rightarrow & (j\ ((\mathsf{lambda}\ (z)\ s_0)\ ())) \\
 & & \parallel \langle (), \emptyset, (\mathsf{setdoc}\, j), \alpha \rangle \parallel \rho_0 \parallel \{j\} \\
 & \overset{*}{\rightarrow} & \mu_0 \parallel \langle q, \mu_1, \emptyset, q \rangle \parallel \rho_1 \\
 & & \text{where } \mu_1 = \{q \mapsto (\alpha, (\langle \mathsf{HTML} \rangle\, p\, r)), p \mapsto (q, (\langle \mathsf{DIV} \rangle\ c_0))\} \\
 & & \qquad\qquad\quad \cup \{r \mapsto (q, (\langle \mathsf{DIV} \rangle\ c_1))\} \\
 & & \qquad\quad c_0 = ((\mathsf{lambda}\ (y)\ y)\ ()) \text{ and } c_1 = ((\mathsf{lambda}\ (x)\ x)\ ()) \\
 & \overset{*}{\rightarrow} & \mu_0 \parallel \langle (), \mu_2, \emptyset, q \rangle \parallel \rho_1 \quad \text{where } \mu_2 = \mu_1[p \mapsto (q, (\langle \mathsf{DIV} \rangle\ \varepsilon))] \cup \{y' \mapsto ()\} \\
 & \overset{*}{\rightarrow} & \mu_0 \parallel \langle (), \mu_3, \emptyset, q \rangle \parallel \rho_1 \quad \text{where } \mu_3 = \mu_2[r \mapsto (q, (\langle \mathsf{DIV} \rangle\ \varepsilon))] \cup \{x' \mapsto ()\}
\end{array}
\begin{array}{l}
(\textsc{Init}) \\
\\
(\ldots \textsc{ServRet1}) \\
\\
\\
\\
(\textsc{Script}, \textsc{AppC}, \textsc{VarC}) \\
(\textsc{Script}, \textsc{AppC}, \textsc{VarC})
\end{array}$$

Figure 12. DOM Extensions: Example 2

[7] J. Franks et al. HTTP Authentication: Basic and Digest Access Authentication. Technical Report RFC 2617, The Internet Society, 1999.

[8] P. Gardner, G. Smith, M.J. Wheelhouse, and U. Zarfaty. DOM: Towards a formal specification. In *Proceedings of the ACM SGIPLAN workshop on Programming Language Technologies for XML (PLAN-X)*, California, USA, January 2008.

[9] P. Gardner, G. Smith, M.J. Wheelhouse, and U. Zarfaty. Local Hoare reasoning about DOM. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 261–270, Vancouver, BC, Canada, June 2008.

[10] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *Automated Software Engineering*, 2004.

[11] Paul Graunke, Robby B. Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling Web Interactions. In *European Symposium on Programming*, Poland, 2003.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthy. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, Slovenia, June 2010.

[13] R. Kelsey, W. Clinger, and J. Rees. The Revised(5) Report on the Algorithmic Language Scheme. *Higher Order and Symbolic Computation (HOSC)*, 11(1), September 1998.

[14] A. Le Hors et al. Document Object Model (DOM) level 2 Core Specification. Technical Report REC-DOM-Level-2-Core-20001113, W3C, November 2000.

[15] F. Loitsch and M. Serrano. *Trends in Functional Programming*, volume 8, chapter 9: Hop Client-Side Compilation, pages 141–158. (M. T. Morazán ed.), Seton Hall University, Intellect Bristol, UK/Chicago, USA, 2008.

[16] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[17] C. Queinnec. The influence of browsers on evaluators. In ACM SIGPLAN *Int'l Conference on Functional Programming (ICFP)*, pages 23–33, Montréal, Canada, September 2000.

[18] M. Serrano. HOP, a fast server for the diffuse web. In *11th international conference on Coordination Models and Languages (COORDINATION)*, LNCS 5521, pages 1–26, Lisbon, Portugal, June 2009.

[19] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the web 2.0. In *Proceedings of the First Dynamic Languages Symposium (DLS)*, Portland, Oregon, USA, October 2006.

[20] M. Serrano and C. Queinnec. A multi-tier semantics for Hop, 2010.